ENCICLOPEDIA PRACTICA DE LA

INFORMATICA APLICADA

34

Ada

AIA

SINCLAIR AMSTRAD COMMODORE IBM



EDICIONES SIGLO CULTURAL

ENCICLOPEDIA PRACTICA DE LA

INFORMATICA APLICADA

34

Ada

EDICIONES SIGLO CULTURAL

Una publicación de

EDICIONES SIGLO CULTURAL, S. A.

Director-editor:
RICARDO ESPAÑOL CRESPO.

Gerente:
ANTONIO G. CUERPO.

Directora de producción:
MARIA LUISA SUAREZ PEREZ.

Directores de la colección:
MANUEL ALFONSECA, Doctor Ingeniero de Telecomunicación y Licenciado en Informática
JOSE ARTECHE, Ingeniero de Telecomunicación

Diseño:
BRAVO-LOFISH.

Dibujos:
JOSE OCHOA Y ANTONIO PERERA.

Tomo XXXIV. **Ada**
AULA DE INFORMATICA APLICADA
ENRIQUE SERRANO, Ingeniero de Telecomunicación

Ediciones Siglo Cultural, S. A.

Dirección, redacción y administración:
Pedro Teixeira, 8-2.ª planta (Ed. Iberia Mart I). Teléf. 810 52 13. 28020 Madrid.

Publicidad:
Gofar Publicidad, S. A. Benito de Castro, 12 bis. 28028 Madrid.

Distribución en España:
COEDIS, S. A. Valencia, 245. Teléf. 215 70 97. 08007 Barcelona.
Delegación en Madrid: Serrano, 165. Teléf. 411 11 48.

Distribución en Ecuador: Muñoz Hnos.

Distribución en Perú: DISELPESA.

Distribución en Chile: Alfa Ltda.

Importador exclusivo Cono Sur:
CADE, S.R.L. Pasaje Sud América, 1532. Teléf.: 21 24 64.
Buenos Aires - 1.290. Argentina.

ISBN del tomo: 84-7688-129-0
ISBN de la obra: 84-7688-018-9.

Fotocomposición:
ARTECOMP, S. A. Albarracín, 50. 28037 Madrid.

Imprime:
MATEU CROMO. PINTO (Madrid).

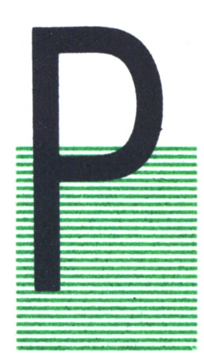
Depósito legal: M. 17.072-1987
Printed in Spain - Impreso en España.

Suscripciones y números atrasados:
Ediciones Siglo Cultural, S. A.
Pedro Teixeira, 8-2.ª planta (Ed. Iberia Mart I). Teléf. 810 52 13. 28020 Madrid.

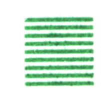
Junio, 1987.
P.V.P. Canarias: 365,-

INDICE

ANTECEDENTES HISTORICOS 1



HISTORIA DEL ADA



PARA comenzar diremos que el lenguaje fue denominado Ada en honor de la primera programadora de la historia, lady Ada Augusta Byron (1816 - 1852), condesa de Lovelace, hija del poeta lord George Byron. Lady Ada colaboró con Charles Babbage en sus trabajos acerca de la «máquina diferencial», predecesora de los modernos ordenadores.

El Departamento de Defensa de los Estados Unidos (Department of Defense, DoD) gasta anualmente miles de millones de dólares en aplicaciones software, incluyendo diseño, desarrollo, adquisición, gestión, soporte operacional y mantenimiento. Parte de estos gastos se emplean en software para los llamados «bembedded computers» o calculadores empotrados, subsistemas que forman parte de sistemas más complejos, como sistemas de comunicaciones para control de tráfico aéreo, sistemas militares, etc.

A comienzos de la década de los años setenta, el DoD inicia un proyecto de estudio de los lenguajes de programación existentes con la idea de estandarizarlos.

En 1975 comienza un programa con el objetivo de estandarizar un lenguaje para aplicaciones en tiempo real. La idea era proveer un único lenguaje de programación común para ordenadores militares. Para ello se creó una comisión que dirigiese los esfuerzos de búsqueda de este lenguaje. A esta comisión se la llamó High Order Language Working Group (HOLWG) (Grupo de Trabajo para Lenguaje de Alto Nivel).

Los razonamientos esgrimidos para la búsqueda de dicho lenguaje común fueron los siguientes:

— La utilización de lenguajes de alto nivel lleva consigo una reducción en los costes de programación,

— Con lenguajes de alto nivel la legibilidad de los programas aumenta, facilitando, por tanto, su modificación y mantenimiento.

— Un lenguaje de alto nivel puede ser utilizado en las fases de especificación, proporcionando una gran ayuda en las fases de verificación y pruebas.

— Si se reduce el número de lenguajes utilizados se consigue un gran beneficio económico.

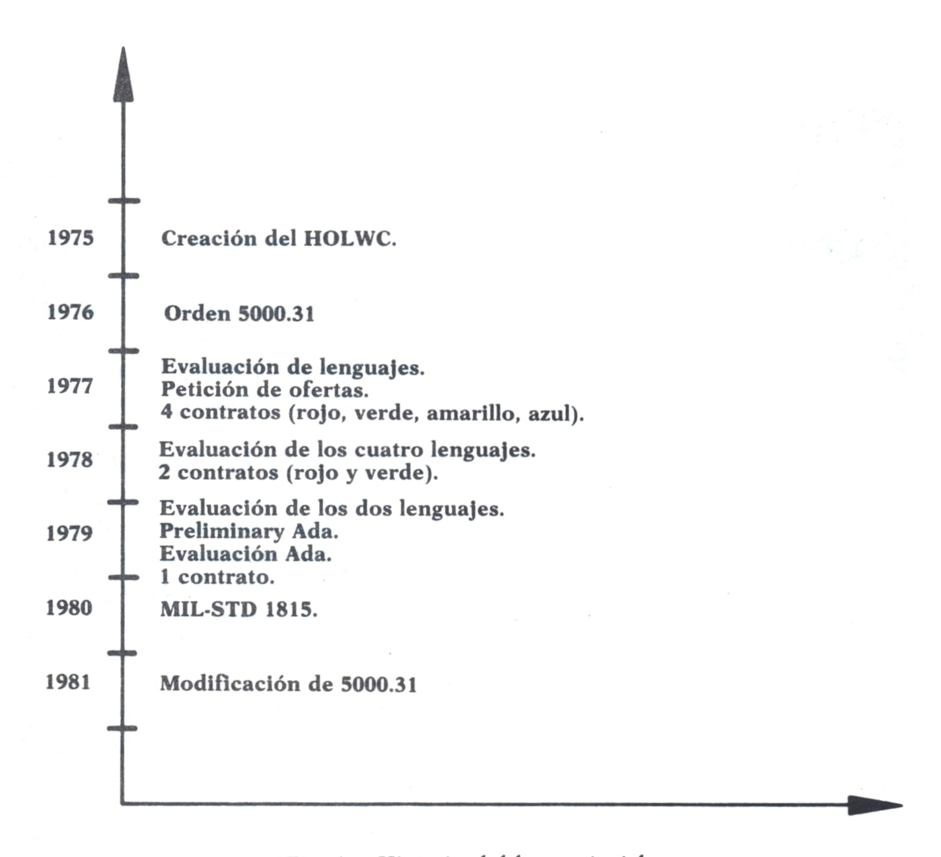


Fig. 1. Historia del lenguaje Ada.

El HOLWG estableció una serie de requerimientos para el nuevo lenguaje común, publicando una serie de documentos que contenían sucesivos refinamientos de sus requisitos. Los documentos elaborados fueron: Strawman (1975), Woodenman (1975), Tinman (1976), Ironman (1977), Revised Ironman (1977), Sandman (1978), Steelman (1978), Pebbleman (1978) y Stoneman (1979).

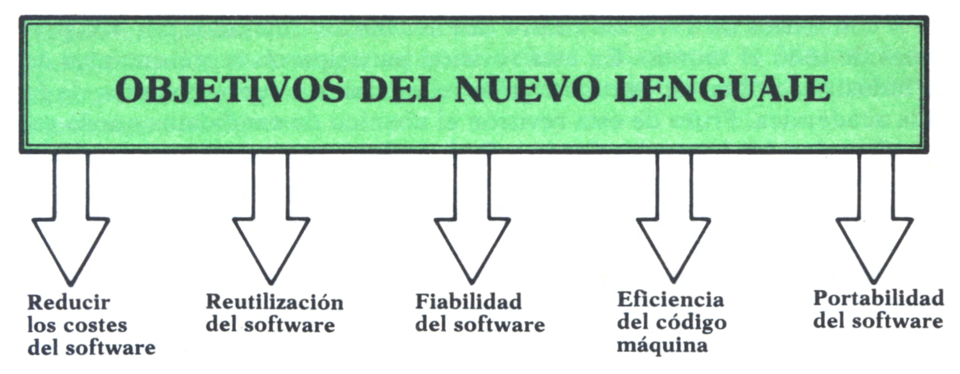


Fig. 2. Objetivos del lenguaje.

Los objetivos que debían ser cubiertos por el nuevo lenguaje eran:

— Reducir los costes del software.

— Facilitar la reutilización de partes del software en proyectos diferentes.

— Aumentar la fiabilidad del software.

— Aumentar la legibilidad de los programas.

— Eficiencia del código máquina.

— Definición completa del lenguaje para evitar la existencia de dialectos.

— Ser un lenguaje «portátil», pudiendo ser utilizado en una gran variedad de máquinas.

A comienzos de 1977, el HOLWG comienza la evaluación de una serie de lenguajes existentes con el propósito de determinar si alguno o una combinación de dichos lenguajes satisfacía los requerimientos del documento Tinman. Alrededor de 26 lenguajes fueron evaluados por distintas organizaciones llegando a la conclusión de que ninguno de los lenguajes podía ser adoptado como lenguaje común y, si bien el desarrollo de éste era posible con la tecnología disponible en el año 1977, el diseño del nuevo lenguaje debía realizarse a partir de uno de los lenguajes existentes, excluyendo a Fortran, Cobol, Tacpol, CMS-2, Jovial J3B y J73, Simula 67, Algol 60 y Coral 66. Se recomendaba, como base partida, Pascal, PL/1 o Algol 68.

En mayo de 1977, el DoD lanza una petición de oferta para producir un prototipo según los requerimientos Ironman, recibiéndose ofertas de firmas nacionales y extranjeras. Obtienen contratos cuatro de ellas para realizar sus lenguajes prototipos.

Las cuatro firmas eran Intermetrics, Cii-Honeywell Bull, SRI International y SofTech. A sus proyectos se les asignan los colores clave Red, Green, Yellow y Blue. Las compañías mencionadas propusieron el lenguaje Pascal como lenguaje de partida.

A comienzos de 1978 los cuatro diseños fueron revisados por 400 personas de todo el mundo. En esta revisión participaron representantes de la industria, militares, agencias gubernamentales y representantes de la vida académica. Fruto de esta revisión el abanico de candidatos quedó reducido a dos: los lenguajes Green y Red. Ambos consiguieron un contrato para un año más de desarrollo, según los requerimientos Steelman. En marzo de 1979, los diseños Green y Red son enviados para su evaluación a distintos centros de desarrollo de software para su evaluación. Una vez valorados los informes se proclama ganador al lenguaje Green. Ada se convierte en el nombre oficial del lenguaje en mayo de 1979.

Los factores que decidieron la victoria del lenguaje Green sobre el Red fueron sus construcciones más avanzadas, su diseño más estable y la compilación separada de unidades de programa.

A continuación se inicia un período de refinamiento del lenguaje, que acaba a mediados del año 1980 con la publicación de un nuevo manual del lenguaje Revised Ada.

Ada constituye actualmente la norma militar MIL - STD 1815, firmada el 10 de diciembre de 1980.

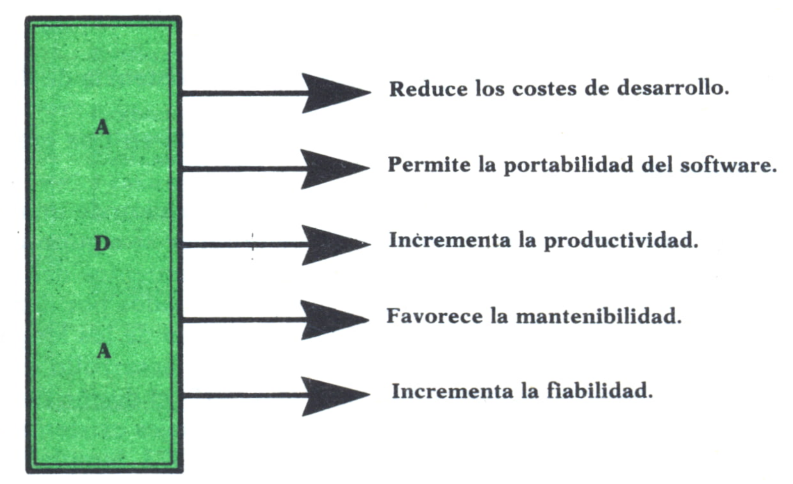
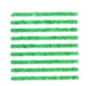


Fig. 3. Aportaciones de Ada.

El HOLWG dejó paso al Comité de Control para el lenguaje Ada (Ada Language Control Board, ALCD) y al Ada Joint Project Office (AJPO). El primero tiene la responsabilidad de velar por la conformidad con el estándar de las futuras realizaciones. En cuanto al segundo, su principal misión es el soporte, distribución y desarrollo de herramientas y librerías, coordinando las actividades del DoD para optimizar la utilización de los diferentes recursos comunes.

Existe, además, otro comité, el Ada Technical Working Group (ATWG), presidido por el DoD responsable de definir y mantener el Ada estándar, y compuesto por representantes del gobierno, industria, servicios públicos e intereses extranjeros.



VALIDACION DEL COMPILADOR ADA

En 1979, el DoD lanza una petición de oferta para proveerse de un Ada Compiler Validation Capability, conjunto de pruebas, procedimientos y documentación elaborados para asegurar el desarrollo de compiladores que cumplan el estándar del lenguaje.

La empresa SoftTech de Waltham, Mass, recibe el encargo de elaboración de la ACVC. El desarrollo de la ACVC comenzó antes de finalizar la definición del estándar Ada.

La ACVC conta de:

— Implementer's Guide (IG).

— Programas de pruebas.

— Herramientas de soporte de validación.

Los programas de prueba comprenderán unos 1.000 programas, a una media de 50 líneas por programa, clasificados en las siguientes categorías:

* *Clase A:* Pruebas que pasan si no se detectan errores al compilar.

* *Clase B:* Son programas ilegales. Pasan si todos los errores que contienen son detectados.

* *Clase L:* Programas ilegales que contienen errores que no pueden ser detectados antes del montaje.

* *Clase C:* Programas ejecutables que se autocomprueban.

* *Clase D:* Pruebas de capacidad.

* *Clase E:* Pruebas de ambigüedad del estándar.



ENTORNO DE SOPORTE A LA PROGRAMACION DE ADA (APSE)

El propósito de un APSE (Ada Programming Support Environment, APSE), es el soporte del desarrollo y mantenimiento del software de aplicaciones Ada a lo largo de su ciclo de vida, con particular énfasis en el software para aplicaciones de «embedded computers». Las características principales de un APSE son:

La base de datos

Contiene toda la información asociada con cada proyecto.

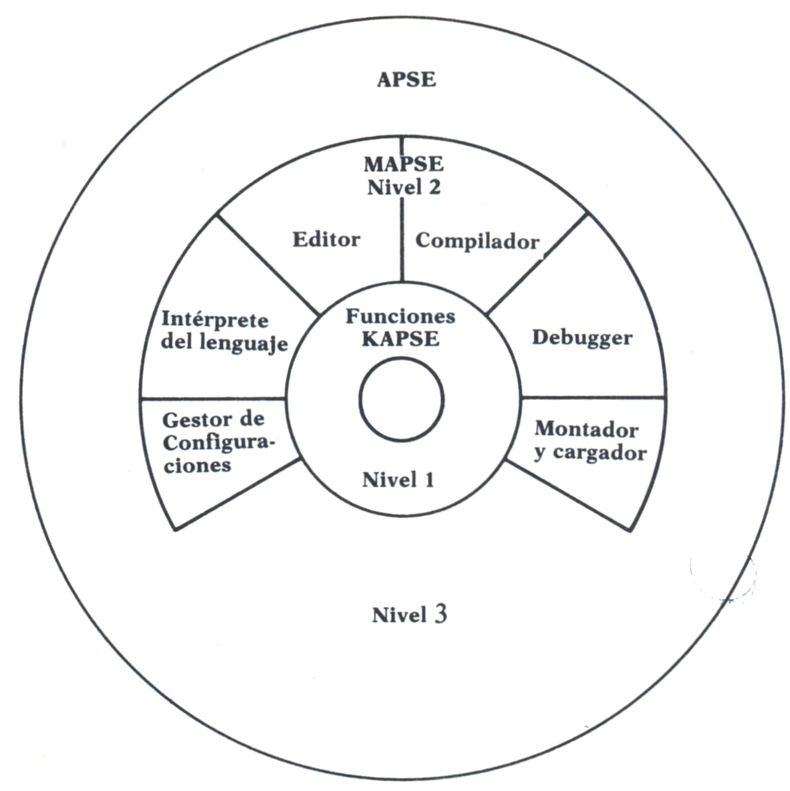


Fig. 4. Estructura de APSE.

Interfases del sistema y usuario

Incluye el lenguaje de control que constituye la interfase con el usuario y las interfases sistema entre la base de datos y el conjunto de herramientas.

El conjunto de herramientas

Compuesto por los útiles para desarrollo de programas, mantenimiento y control de configuración.

Stoneman indica los siguientes niveles de un APSE:

* *Nivel 0:*

Hardware soporte y software apropiado.

* *Nivel 1:*

KAPSE, contiene las funciones de la base de datos, comunicación y soporte de ejecución que permiten la ejecución de un programa Ada.

* *Nivel 2:*

MAPSE. Provee un conjunto mínimo de útiles escritos en Ada y soportados por el KAPSE.

* *Nivel 3:*

APSE. Construido por extensiones del MAPSE para dar soporte completo a aplicaciones o metodologías particulares.



REALIZACIONES EN EE.UU.

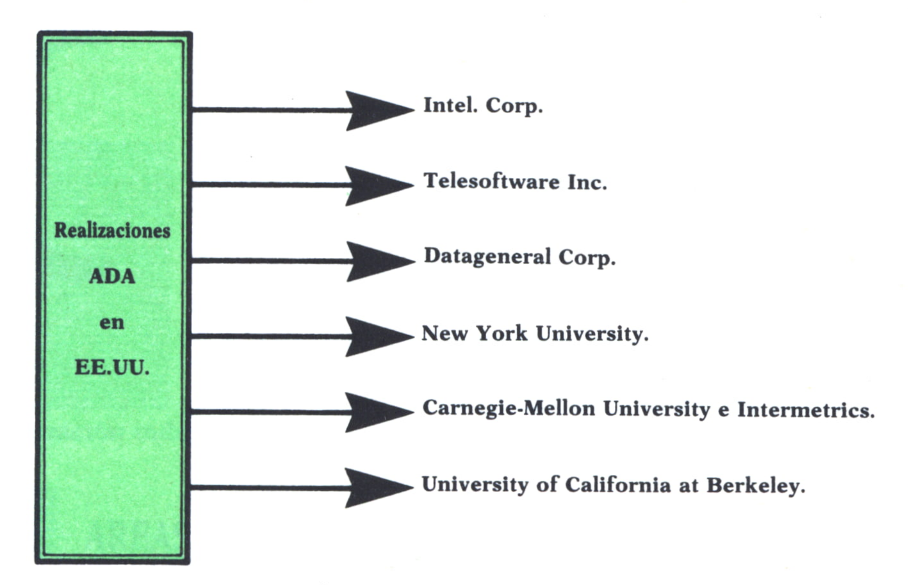


Fig. 5. Realizaciones Ada en EE.UU.

Empresas que de una forma u otra tienen relación con el lenguaje de programación Ada son:

* *Data General Corp.*, dedicada al desarrollo de compiladores.

* *New York University*, dedicada al desarrollo de compiladores Ada con fines educacionales.

* *Carnegie Mellon University e Intermetrics*, dedicadas a desarrollar compiladores Ada, bajo contrato de la DARPA (Agencia de Investigaciones Avanzadas para la Defensa).

* *Telesoftware Inc.*, filial de la Western Digital Corp. que produce el Pascal.

* *Intel Corp.*, adopta el lenguaje Ada para el microprocesador que tenía en desarrollo, el APX432. El compilador de Intel sería un compilador cruzado (cross compiler), funcionando en VAX y se preveía su aparición en 1981.

CARACTERISTICAS DEL ADA 



COMO se vio en el capítulo anterior los objetivos para conseguir el lenguaje Ada se han orientado a desarrollar un lenguaje común, así como un conjunto de herramientas que faciliten su utilización en sistemas militares, fundamentalmente, con misiones críticas. Pero el lenguaje Ada no queda reducido al ámbito de los sistemas de armas, ya que posee una amplia variedad de usos para aplicaciones informáticas en tiempo real. Ejemplos de sistemas para los cuales el entorno Ada resulta de gran interés son la automatización industrial, control de procesos, etc.



AREAS DE APLICACION DEL ADA

La siguiente lista intenta dar al lector una amplia visión de los campos de utilización del lenguaje Ada.

* Aplicaciones en tiempo real, cálculo numérico, procesamiento de señal y control de hardware:

— Sistemas de comunicaciones.
— Sistemas de navegación.
— Sistemas de control para robótica.
— Sistemas de control para procesos químicos.
— Sistemas de control para plantas nucleares.
— Sistemas de control para tráfico aéreo.
— Sistemas de radar, sonar y óptica.

* Sistemas para la gestión de datos:

— Sistemas para la gestión de base de datos.
— Sistemas para la gestión de información.

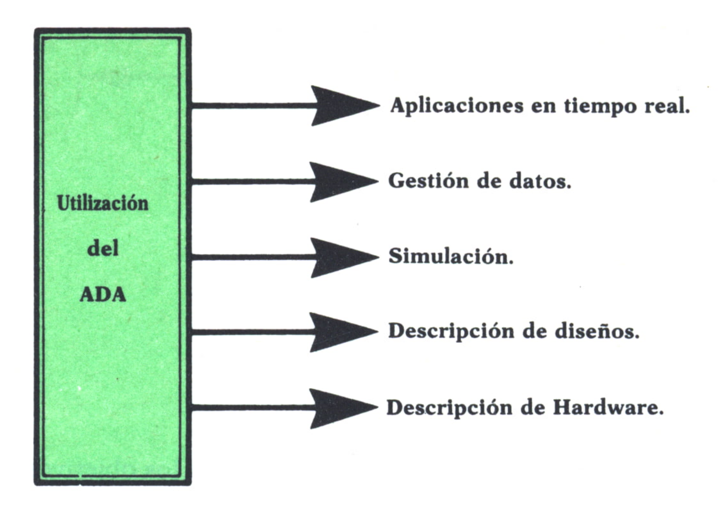


Fig. 1. Aplicaciones del Ada.

* Sistemas para la simulación (simuladores):

— Sistemas para juegos de guerra.
— Simuladores de vuelo.
— Entrenadores.
— Prototipos.
— Sistemas gráficos.

* Descripción de diseños:

— Lenguajes para el diseño de sistemas.

* Descripción de hardware para circuitos de alta escala de integración (VLSI):

— Lenguajes para descripción de hardware.
— Lenguajes para la modelización del hardware y la simulación.

Como podemos apreciar por la extensa lista de aplicaciones del Ada éste es, con diferencia, el lenguaje que mayor número de sistemas pueden utilizarlo hoy día.



CARACTERISTICAS DEL ADA

El Ada, como todo lenguaje, tiene sus partidarios y detractores. Estos últimos achacan al lenguaje una gran complejidad, apto tan sólo para programadores experimentados.



Fig. 2. Características de programas Ada.

Ada posee una gran variedad de características. Todos los programas en Ada se componen de una o más unidades de programa que pueden ser compiladas independientemente. Un programa Ada puede utilizar características del lenguaje que permiten la construcción de un programa, a partir de una colección de *unidades de programa en librería*. Estas unidades de programa pueden estar constituidas por módulos definidos por el usuario, unidades predefinidas en el lenguaje y unidades definidas en el entorno. Estas unidades de programa pueden incluir una combinación de las siguientes entidades:

* *Subprogramas (funciones o procedimientos).*

* *Paquetes (<<Packages>>).*
* *Tareas (<<Task>>).*
* *Genéricos (<<Generics>>).*

Además, cada una de estas unidades de programa están constituidas por dos partes:

— *Especificación (<<specification>>)*

Esta parte de la unidad de programa contiene la información que necesitan otras unidades de programa.

— *Cuerpo (<<body>>)*

Esta parte de la unidad de programa contiene los detalles de la realización de la unidad.

Las características básicas de las unidades de programa son las siguientes:

* Los subprogramas pueden venir expresados de dos maneras: procedimientos y funciones.
* Los subprogramas pueden tener parámetros que permitan que la información pueda ser pasada a y desde el subprograma al punto de llamada.
* Los procedimientos, en general, serán utilizados para indicar una serie de acciones.
* Las funciones, en general, serán usadas para el tratamiento y la devolución de una variable.
* Los paquetes (<<packages>>) pueden ser utilizados para definir

un grupo común de datos o tipos de datos, un conjunto de unidades de programa relacionados, o un tipo abstracto de datos (tad).

* Los paquetes permitirán al programador especificar qué parte de la unidad será visible al usuario del <<package>> y qué otra parte será oculta.

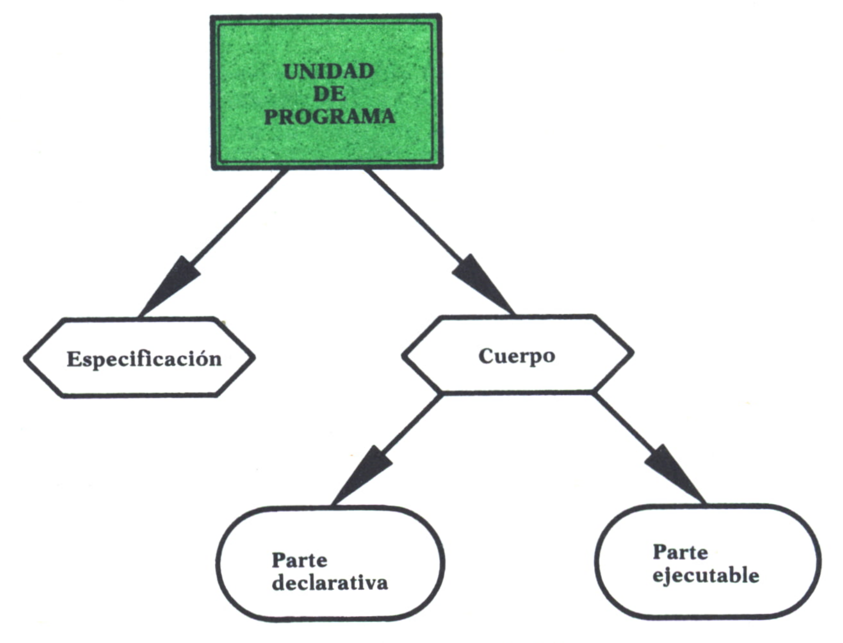


Fig. 3. Composición de una unidad de programa.

* Las tareas (< <tasks> >) pueden ejecutarse en paralelo.
* Las especificaciones y los cuerpos (<<bodies>>) pueden compilarse por separado.
* El cuerpo de las unidades de programa se compone de dos partes: *una parte declarativa,* la cual define las entidades que serán utilizadas en la unidad de programa, y una *parte ejecutable,* la cual define la secuencia de ejecución de las instrucciones de la unidad de programa.
* El cuerpo de una unidad de programa puede tener asociado un conjunto de *tratamientos de excepción* (< <exception handlers> >) que pueden ser utilizados para la recuperación de condiciones de error u otros requisitos de tratamientos de error que requiera la aplicación.

También debemos saber que la parte declarativa de una unidad de programa asocia un nombre con la entidad declarada que la unidad de programa utilizará. Estas entidades pueden incluir un tipo de datos, una constante, una variable, un subprograma anidado, una unidad de <<package>> anidada, una unidad de tarea anidada o una unidad genérica.

La parte ejecutable de una unidad de programa está compuesta de una combinación de los siguientes tipos de instrucciones:

— *Assignment.*

Instrucción de asignación. Una variable pasa a tener el valor computado de una expresión.

— *Procedure call.*

Instrucción de llamada a un procedimiento. Con ésta invocaremos la ejecución de un subprograma.

— *Case.*

Instrucción de selección. Los amantes del Pascal reconocerán en esta instrucción una estructura de control de selección que permite listar distintas alternativas y elegir que se ejecute una en tiempo de ejecución. La selección se hace comparando el valor de una expresión con un rótulo asignado a cada acción.

— *If.*

Instrucción condicional. Permite la selección de un conjunto de instrucciones dependiendo de una condición.

— *Loop.*

Instrucción de iteración. Es una sentencia de control que permite la ejecución de un bloque de instrucciones de forma iterativa mientras se dé una condición.

— *Exit.*

Instrucción de salida. Permitirá abandonar un bucle (<<loop>>) de manera condicional o incondicional.

— *Block.*

Permite la introducción de entidades locales que aplican a la secuencia de instrucciones contenidas en el bloque.

— *Raise.*

Genera una condición de error definida por el programador.

Ada posee una amplia variedad de tipos de datos. Cada objeto en el lenguaje Ada, constante o variable, tiene un *tipo de datos* que especifica el conjunto de valores permitidos para dicho objeto, así como el conjunto de operaciones que pueden desarrollarse sobre ellos. Los tipos de datos soportados por Ada son:

— *Tipos enteros.*
— *Tipos en coma fija.*

— *Tipos en coma flotante.*
— *Tipos enumerados.*
— *Tipos array.*
— *Tipos registro.*
— *Tipos acceso (punteros).*
— *Tipos privados (<<privates>>).*

En relación con los tipos de datos, Ada permite al programador especificar diferentes características de los tipos de datos, en tiempo de ejecución, tales como direcciones, número de bits, etc., a través de la utilización de cláusulas de representación (< <representation clauses> >).

En capítulos posteriores se realizará una exposición más detallada de las características del lenguaje.



ESTUDIO COMPARATIVO DE ADA CON OTROS LENGUAJES

En ocasiones puede resultar interesante resaltar las peculiaridades de un lenguaje, comparado con otros, con el fin de tener una idea más completa de dicho lenguaje.

Se han elegido en esta ocasión tres lenguajes populares tales como el Pascal, Fortran y C. La tabla que presentamos a continuación está basada en los trabajos de J. V. Cugini, de la National Bureau of Standards, el cual analizó varios lenguajes con el objeto de proporcionar una guía en la selección y utilización de tales lenguajes.

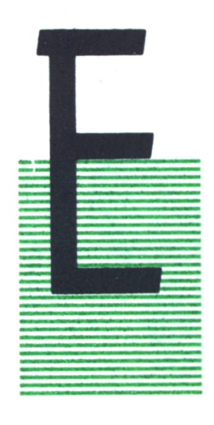
Características del lenguaje	**ADA**	**C**	**FORTRAN**	**PASCAL**
Estilo sintáctico				
Formato libre	Sí	Sí	No	Sí
Formas de etiqueta	Nombre	Nombre	Número	Número
Tamaño máx. identificador	Ll (**)	8	6	Ll (**)
Variables no declaradas	No	No	Sí	No
Control de ejecución				
Programación estructurada	Sí	Sí	Parcial	Sí
Bloques	Sí	Sí	No	Sí
Recursividad	Sí	Sí	No	Sí
Procedim. genéricos	Sí	No	No	No
Tratamiento excepciones	Sí	No	No	No
Tratamiento de errores	Sí	No	No	No
Concurrencia	Sí	No	No	No

(**) NOTA: Ll = Longitud de línea.

Características del lenguaje	ADA	C	FORTRAN	PASCAL
Control de datos				
Variables automáticas	Sí	Sí	Sí	Sí
Variables estáticas	No	Sí	Sí	No
Variables asignadas por el usuario	Sí	Sí	No	Sí
Datos externos	Sí	Sí	Sí	No
Tipos definidos por el usuario	Sí	Sí	No	Sí
Comprobación de tipos	Sí	No	No	Sí
Operadores definidos por el usuario	Sí	No	No	No
Grado de flexibilidad	Alto	Medio	Bajo	Medio
Datos numéricos				
Enteros	Sí	Sí	Sí	Sí
Coma fija	Sí	No	No	No
Coma flotante	Sí	Sí	Sí	Sí
Complejos	No	No	Sí	No
Operadores básicos	Sí	Sí	Sí	Sí
Funciones numéricas	Pocas	Ninguna	Varias	Algunas
Librerías numéricas	Pocas	Algunas	Varias	Algunas
Arrays				
Dimensiones	Ilimitado	Ilimitado	7	Ilimitado
Tipo de subíndice	Discreto	Entero	Entero	Discreto
Límite inferior definido por el usuario	Sí	No	Sí	Sí
Asignación de array	Sí	No	No	Sí
Inicialización de array	Sí	No	Sí	No
Comparación de array	Sí	No	No	No
Registros				
Registros/Estructuras	Sí	Sí	No	Sí
Asignación de registro	Sí	No	No	Sí
Comparación de registro	Sí	No	No	No
Formato variab./dinámico	Sí	No	No	Sí
Otros tipos de datos				
Carácter	Sí	Sí	Sí	Sí
Cadena	Sí	Sí	Sí	Sí
Lógico/boolean	Sí	Sí	Sí	Sí
Enumerado	Sí	No	No	Sí
Puntero	Sí	Sí	No	Sí
Conjuntos	No	No	No	Sí
Control interno de representación	Sí	No	No	No
Entrada/salida y archivos				
Archivos secuenciales	Sí	Sí	Sí	
Archivos de acceso directo	Sí	Sí	Sí	No
Otros factores				
Bajo nivel de E/S	Sí	No	No	No

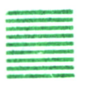
Características del lenguaje	**ADA**	**C**	**FORTRAN**	**PASCAL**
Area de aplicación				
Negocios	Bajo	Bajo	Bajo	Bajo
Cálculo matemático	Moderado	Bajo	Alto	Moderado
Sistemas operativos	Alto	Alto	Bajo	Bajo
Tiempo real	Alto	Bajo	Bajo	Bajo
Enseñanza	Moderado	Bajo	Moderado	Alto
Desarrollos en equipo	Alto	Bajo	Bajo	Bajo
Fiabilidad	Alto	Bajo	Bajo	Moderado
Portabilidad	Alto	Moderado	Alto	Alto
Eficiencia de ejecución	Moderado	Alto	Alto	Moderado
Desarrollos de grandes aplicaciones	Alto	Moderado	Bajo	Bajo

METODOLOGIAS DE DISEÑO ORIENTADAS AL ADA 3



EN este capítulo describiremos, de forma breve, varias metodologías asociadas con el diseño y realización de una aplicación en Ada. También nos referiremos al grado de soporte que proporciona cada una de las metodologías de diseño orientadas a este lenguaje.

También daremos al final de este capítulo una breve descripción de áreas de aplicación de este potente lenguaje, así como las futuras orientaciones de éste.



PRINCIPIOS DE DISEÑO E IMPLEMENTACION

En general, los principios asociados con la ingeniería del software, el diseño y la implementación podemos resumirlos en:

Estructuración

Conjuntos y subconjuntos relacionados forman un todo. Los objetivos de la estructuración en el diseño y la implementación son:

— Reducir la complejidad
— Incrementar la inteligibilidad.

Modularidad

Una forma racional de conseguir la estructuración. Los objetivos de la modularidad son:

— Hacer explícitas las interfases entre partes de la estructura.

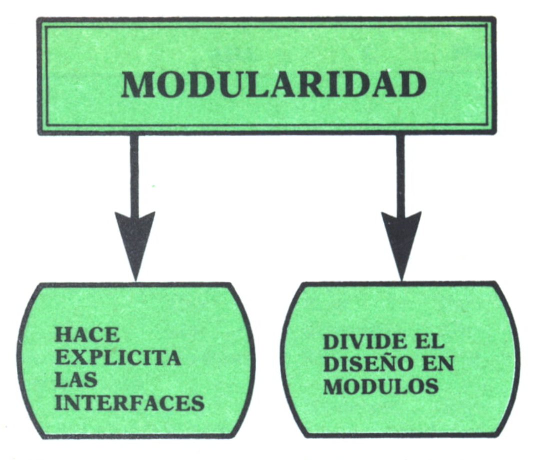


Fig. 1. Características de la modularidad.

— Dividir el diseño en módulos que serán desarrollados independientemente.

Abstracción

Eliminación controlada de detalles, resaltando los esenciales y eliminando los no esenciales. La abstracción ayuda a entender los aspectos de interés, ya que controlamos el nivel de detalle que necesitamos ver.

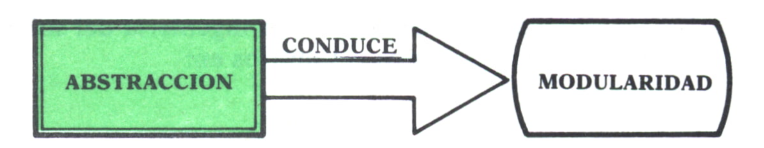


Fig. 2. Características de la abstracción.

Los objetivos de la abstracción, relativos al diseño y la implementación, pueden considerarse similares a los de la modularidad, ya que en algunas metodologías la abstracción se utiliza como el principal criterio para obtener la modularidad.

Ocultamiento

Hace que ciertos detalles o criterios de diseño sean inaccesibles. El ocultamiento, como la abstracción, podría verse como un medio para conseguir la modularidad.

El ocultamiento nos proporciona un método para definir qué partes serían visibles y qué ocultaríamos dentro del módulo. Los detalles que ocultaríamos se basarían en:

— Probabilidad de cambio para un detalle. Si tal probabilidad existe, ocultar un detalle dentro de un módulo minimizará el efecto del cambio.

— Complejidad del detalle.

En resumen, los principios en los que se basan la ingeniería del software, tales como *estructuración, modularidad* y *abstracción* están orientados a reducir la complejidad y, por tanto, aumentarán nuestro conocimiento acerca del diseño y la implementación. Por otra parte, los principios de *modularidad* y *ocultamiento* están orientados a limitar el impacto de los cambios sobre el diseño o la implementación.



Fig. 3. Principios en los que descansa la I. del software.



METODOLOGIAS DE DISEÑO

Como aproximación al diseño de una aplicación describiremos una serie de metodologías que, si bien pueden no satisfacer las más rígidas definiciones y aspectos de lo que significa una metodología, pueden cubrir las necesidades del diseñador de aplicaciones.

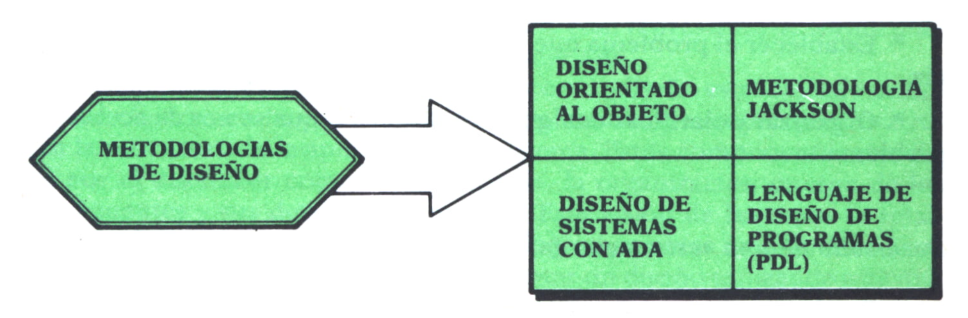


Fig. 4. Metodologías de diseño.

Diseño orientado al objeto

El diseño orientado al objeto es una metodología de diseño descrita por G. Booch y desarrollada por varias personas. Esta metodología, como un proceso de diseño, se concentra sobre la estructura de una aplicación tal y como uno la vería si únicamente observó los objetos abstractos que componen la aplicación y las operaciones sobre esos objetos. Concentrarnos sobre dichos objetos se aparta del punto de vista más tradicional que apunta a que toda estructura de una aplicación derivaría de una descomposición «top-down» (de arriba abajo) de las funciones que deben ser desarrolladas por esa aplicación. El *diseño orientado al objeto* puede aplicarse de una manera «top-down». En este caso, esta metodología seguiría el camino natural, en el cual los objetos y los datos de la aplicación se descomponen en lugar del camino en el cual se dividen las funciones.

Esta metodología puede utilizarse con aplicaciones de menos de 30.000 líneas de código, si bien puede ser empleada para proyectos de mayor envergadura, siempre que se utilice en conjunción con otra u otras metodologías que permitan dividir la aplicación en pequeños trozos, a los cuales se les pueda aplicar, de forma independiente, la metodología que estamos analizando.

La metodología del *diseño orientado al diseño* puede ser vista como una metodología de diseño intermedia entre el diseño preliminar y el diseño detallado. La principal ventaja de esta metodología, de cara al diseñador de aplicaciones en fase de iniciación, es que es «intuitiva» y puede ser utilizada de forma sencilla para un eficaz aprovechamiento de las características del Ada.

La metodología del *diseño orientado al objeto* se construye basándose en un proceso de tres etapas, las cuales identificamos a continuación:

Definición del problema

* Establecer el problema que deseamos solucionar en una sola sentencia.

* Organizar y clarificar cualquier información que se nos dé acerca del problema.

Desarrollo de una «estrategia» informal para el dominio del problema

* Establecer en un único párrafo «una solución» al problema mencionado en líneas anteriores.

Formalización de la estrategia

* Desarrollar la estrategia informal para un diseño estructural en Ada:

1. Identificando los objetos de la aplicación y sus atributos.
2. Identificando las operaciones sobre esos objetos.
3. Estableciendo las interfases para los objetos y operaciones.
4. Implementando las operaciones en Ada.

Las representaciones utilizadas por esta metodología de diseño incluye notaciones gráficas para expresar la estructura y las interacciones entre elementos del diseño y una completa sintaxis del lenguaje de diseño de programas Ada. Esta notación gráfica fue desarrollada para resaltar las interfases de los elementos que compondrán el diseño.

Metodología JSD

La metodología JSD (Jackson System Development) fue desarrollada por M. Jackson sobre métodos de análisis de requisitos, diseño y programación durante la década de los setenta. JSD no es una metodología «top-down» como eran ciertas metodologías de esa época. JSD está basada en la observación de que la estructura de un programa refleja la estructura de los datos, es decir, el diseño es un proceso de exposición de la estructura de los datos inherentes en el problema y transformación de esa estructura de datos dentro de una estructura de programa o sistema.

La metodología Jackson es utilizada en Europa y se emplea para desarrollo de sistemas de gestión, y a veces ha sido utilizada para algunos sistemas en tiempo real.

Diseño de sistemas con Ada

Esta metodología proporciona una aproximación «top-down» al aspecto arquitectónico del diseño de una aplicación Ada. Esto se consigue con el uso de una notación gráfica del diseño, la cual representa, de forma muy aproximada, las características del lenguaje Ada.

La metodología ha sido caracterizada como «diseño estructurado orientado al objeto», en la cual los objetos son los componentes que desarrollan la estructura de la aplicación o el sistema. Estos componentes son vistos por el diseñador como «cajas negras», la organización interna de los componentes y sus detalles son ocultos. Esta visión de las cosas es totalmente coherente con las «unidades de programa» vistas en Ada. Esto hace de Ada un lenguaje natural para implementar un diseño utilizando esta metodología.

Como Ada soporta el anidamiento de estructuras de programa, podría-

mos esperar que cualquier metodología de diseño permitiría desarrollar el diseño de una manera «top-down» usando un refinamiento progresivo. El diseño de sistemas con la metodología Ada soporta esto de una forma directa, y permite al diseñador crear un conjunto de «subsistemas» que pueden representar a la aplicación como una estructura única. A partir de aquí pueden crearse subsistemas adicionales que representen al nivel superior de diseño. Este proceso puede repetirse para cada uno de los nuevos niveles hasta que el diseñador ha especificado completamente el diseño.

El diseño de sistemas con metodología Ada no propone un proceso de diseño formal de una manera particular, pero proporciona al diseñador un esquema de representación y una guía en la aplicación del Ada a los problemas del mundo real.

Los procesos de diseños, a los cuales esta metodología parece adaptarse, son:

— Diseño orientado al objeto.
— Diseño estructurado.

Lenguaje de diseño de programas

El lenguaje de diseño de programas (Program Design Language, PDL), también llamado «seudocódigo» por algunos autores, proporciona un medio para la representación de datos y procesos a través de un lenguaje textual. Este lenguaje debe ser lo suficientemente preciso de manera que nos permita describir el software y expresar un diseño a través de él. Un PDL tiene una sintaxis formal que nos indica las construcciones de procesos y datos, a la vez que incorpora descripciones del lenguaje natural en un formato libre para explicar ciertos detalles. Si bien un PDL no es directamente ejecutable, técnicas automatizadas para evaluación de diseños han sido desarrolladas por Hitachi, IBM y otros. Las instrucciones de un PDL pueden ser agrupadas en tres categorías:

— Declaración de datos.
— Descripción del proceso.
— I/O (entrada / salida).

Construcciones especiales pueden ser incorporadas, al tiempo que para cierto tipo de aplicaciones particulares pueden desarrollarse nuevas instrucciones, tales como instrucciones para aplicaciones multitarea o en tiempo real.

Con Ada tenemos un lenguaje de programación de propósito general que es lo suficientemente expresivo como para describir todos los aspectos del diseño software. Un lenguaje de diseño de programas, que soporte

todos los tipos de aplicación que deseamos conseguir a través de Ada, debe reunir las siguientes características:

— Abstracción.
— Descomposición.
— Ocultamiento de la información.
— Refinamiento sucesivo.
— Modularidad.

La mayoría de estas características formaron parte de los requisitos establecidos para la creación de Ada, lo cual hace de Ada un lenguaje muy conveniente para un PDL. Para dar un soporte a estos principios, un PDL debe documentar el diseño resultante y debe poseer las siguientes cualidades:

— Debe ser legible e inteligible por las personas que llevan a cabo el proceso de diseño.

— Debe dar una representación completa, concisa, precisa y estructurada del diseño.

— Debe ser verificable y troceable, con relación a los requisitos de diseño establecidos para el sistema.

Como ejemplo de una descripción de lenguaje de diseño para un trozo de clasificación (SORT) podría ser la siguiente:

IF
El primer elemento > segundo elemento de la lista.
THEN
Intercambia ambos elementos.
ELSE
Reemplaza el primer elemento con el buffer de entrada 1 reemplaza el segundo elemento con el buffer de entrada 2.
ENDIF



PRINCIPIOS SOPORTADOS POR LAS METODOLOGIAS DE DISEÑO ORIENTADAS A ADA

La siguiente tabla resume el grado de soporte que las metodologías descritas en el apartado anterior proporcionan a la ingeniería del software.

Metodología	Características	Grado de apoyo
Diseño orientado al objeto	Estructuración	Alta
	Abstracción	Alta
	Modularidad	Alta
	Ocultamiento	Moderado

Metodología	Características	Grado de apoyo
Metodología Jackson	Estructuración	Alta
	Abstracción	Moderada
	Modularidad	Moderada
	Ocultamiento	Bajo
Diseño de sistemas en Ada	Estructuración	Alta
	Abstracción	Alta
	Modularidad	Alta
	Ocultamiento	Alto
Lenguaje de diseño de programas	Estructuración	Alta
	Abstracción	Moderada
	Modularidad	Moderada

OTRAS AREAS DE UTILIZACION DEL ADA PARA EL FUTURO

Debido a su gran potencia el espectro de utilización de este lenguaje se extiende desde la descripción hardware a la definición de base de datos. Este rango de utilización de Ada puede tomar las siguientes formas:

— Complementar el lenguaje Ada con características especializadas utilizando el concepto de «paquete» (*package*) de Ada. Esta forma ha sido utilizada para ampliar el campo de Ada dentro de nuevos dominios de aplicación.

— Añadir características a Ada con extensiones a su sintaxis y semántica. Esta forma da origen a un nuevo lenguaje, el cual es, a menudo, preprocesado.

— Utilizar conceptos y características selectivas de Ada para un nuevo lenguaje.

Ada es un lenguaje adecuado para aplicaciones de simulación, ya que soporta «procesos concurrentes». Esto permite a Ada ser utilizado como lenguaje de simulación proporcionando una capacidad similar a lenguajes como SIMSCRIPT II.5.

Las características necesarias que sirvan de base para la utilización de Ada como lenguaje de simulación pueden proporcionarse a través de un conjunto de «paquetes» que posean capacidades especiales.

Características propias del Ada, tales como «paquetes», «tareas» (*tasking*) y «genéricos» (*generics*), así como también conceptos derivados de la ingeniería del software, hacen que el desarrollo de «simulaciones» sea, como mínimo, tan sencillo como utilizar lenguajes de simulación especializados.

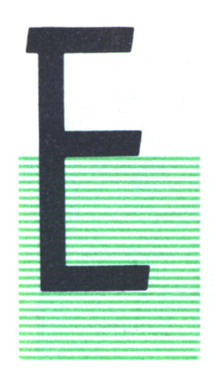
Otra importante aplicación del Ada reside en contemplar a este lenguaje como lenguaje orientado a base de datos. Grandes sistemas, en uso hoy día, requieren no solamente la utilización de un lenguaje de programación de propósito general, sino también necesitan un completo sistema de gestión de base de datos.

Para apoyar esta clase de aplicación, varios grupos se han encargado de la investigación de un lenguaje de aplicación a Ada al que la Computer Corporation of America ha denominado ADAPLEX. Este lenguaje aporta a Ada, a nivel de declaración e instrucción, la definición de datos y capacidad de manipulación propias de un lenguaje interactivo. Para llegar a Adaplex se añadió a Ada un sublenguaje de base de datos, el cual permitiría al codificador de aplicaciones libertad para entremezclar las capacidades de los dos lenguajes de una forma fuertemente integrada, creando un entorno general de lenguaje fácil de usar y mantener. Adaplex ha añadido dos categorías de características a Ada, las cuales se denominan «declaraciones de esquema» e «instrucciones de transacción» Esto puede verse como una extensión al propio lenguaje Ada, que da lugar a un nuevo lenguaje.

Una base de datos en Adaplex tiene características de «interface» similares a un «paquete» de Ada, pero se le ha incorporado como una unidad estructural separada. Dentro de los distintos tipos de «unidades de programa» existentes en Ada, podemos utilizar libremente distintos tipos de instrucciones y expresiones que usan atributos asociados con entidades de la base de datos.

Finalmente, diremos que con la llegada de los circuitos integrados a gran escala (Very Large Integrated Circuit, VLSI), Ada puede ser considerado como un lenguaje de descripción de hardware. El DoD (Department of Defense) se ha intereseado en la aplicación de los circuitos integrados a gran escala para sus sistemas, debido a que este tipo de circuitos aumenta la fiabilidad y reduce los costes. En 1980 el DoD lanzó un programa denominado «Very High Speed Integrated Circuits» (VHSIC) donde se recogen las necesidades del DoD en este área, así como las herramientas necesarias para soportar el desarrollo y uso de dispositivos en tiempo real. Como parte de este programa se intentó definir un lenguaje de descripción de hardware orientado a VHSIC y conocido por las siglas VHDL. Los requisitos para el VHDL establecían que el lenguaje debía soportar el diseño, la documentación y una simulación eficaz del hardware desde el nivel de diseño del sistema hasta el nivel de «puerta»; el lenguaje debía ser extensible a un número adecuado de tecnologías de hardware, metodologías de diseño y entornos de soporte; el lenguaje debía usar construcciones de Ada si esto era posible.

VISION GENERAL DE ADA COMO LENGUAJE 4



EN este capítulo resaltaremos las principales ideas que sirven de base al lenguaje de programación Ada y las compararemos con aquellas otras que sustentan al lenguaje de programación Pascal, con el propósito de establecer los puntos comunes que existen entre ambos. También daremos una breve descripción de aquellas características de Ada que apoyan los principios de la ingeniería del software.



CONSTRUCCIONES

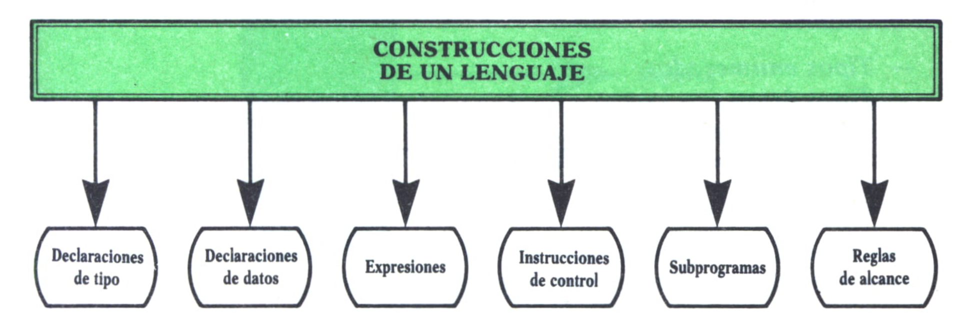


Fig. 1. Construcciones de un lenguaje.

Las construcciones tradicionales de un lenguaje pueden ser clasificadas dentro de las siguientes categorías:

— *Declaraciones de tipo.*
— *Declaraciones de datos.*
— *Expresiones.*

— *Instrucciones de control.*
— *Subprogramas y parámetros.*
— *Reglas de alcance.*

Daremos una breve descripción de cada una de estas categorías y compararemos, en ciertos momentos, las características que proporcionan ambos lenguajes, Ada y Pascal.



DECLARACIONES DE TIPO

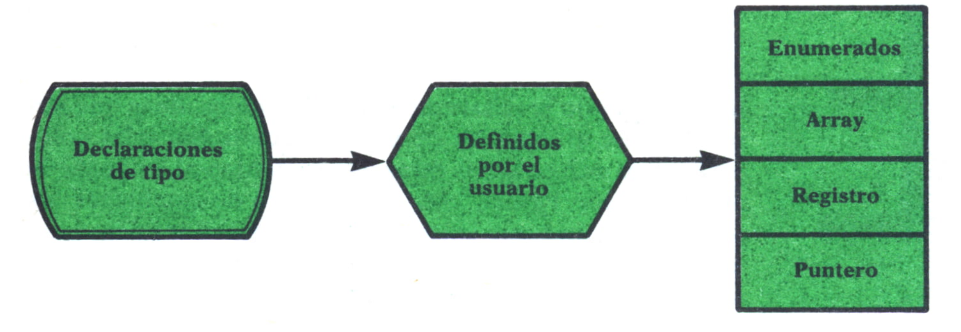


Fig. 2. Declaraciones de tipo.

Tanto Pascal como Ada proporcionan cuatro clases de tipos definidos por el usuario:

— *Tipos enumerados.*
— *Tipos Array.*
— *Tipos registro.*
— *Tipos puntero (llamados tipos acceso* en Ada).

Veamos algunos ejemplos.

type dirección = (Norte, Sur, Este, Oeste); (PASCAL)
type direccion *is* (Norte, Sur, Este, Oeste); (ADA)
type celda = *array* [1..10] *of* integer; (PASCAL)
type celda *is array* (1..10) *of* INTEGER; (ADA)

La palabra reservada *type* se utiliza una sola vez en Pascal para definir los tipos; sin embargo, en Ada es usada para cada declaración de tipo.

Entre los tipos definidos por el usuario, dados anteriormente, no mencionamos el tipo subrango, porque este concepto no corresponde a un tipo en Ada. Una declaración de un subtipo escalar no introduce un nuevo tipo, pero define un conjunto de valores contiguos al tipo base.

type diasemanal = Lunes... Jueves; (PASCAL)

subtype diasemanal *is* dia *range* Lunes... Jueves; (ADA)

Ada proporciona otras dos clases de *subtipos: subtipos array* y *subtipos registro*. Inicialmente pueden declararse subtipos array sin fijar los límites del índice, pero posteriormente debe establecerse el límite a través de una declaración *subtipo*.

En Ada las funciones predefinidas asociadas con tipos se denominan *ATRIBUTOS* del tipo. El atributo *POS* corresponde a la función de Pascal *ord*, y los atributos *PRED* y *SUCC* corresponden en Pascal a las funciones *pred* y *succ*.

Ada posee un importantísimo tipo que no posee Pascal. Este tipo se llama *tipo tarea* (task type) y está relacionado con la <<multitarea>>.



DECLARACIONES DE DATOS

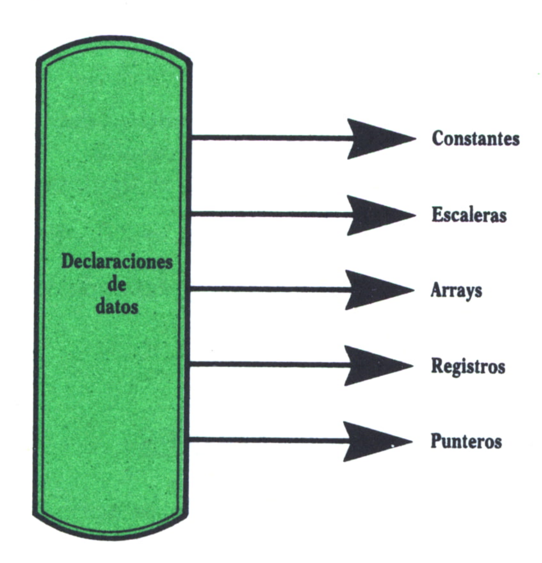


Fig. 3. Declaraciones de datos.

El lenguaje Ada, así como el Pascal, poseen las siguientes clases de declaraciones:

— Declaraciones de constantes.
— Declaraciones de escalares.
— Declaraciones de arrays.
— Declaraciones de registros.
— Declaraciones de punteros.

Las declaraciones de datos en Ada son dinámicas, lo que no ocurre en Pascal, que son estáticas, es decir, en Ada las declaraciones pueden ser expresadas en términos de variables de programa, mientras que en Pascal las declaraciones se expresan en términos de constantes que son conocidas en tiempo de compilación.

En Ada un dato puede ser inicializado en la declaración.

Veamos algunos ejemplos.

const pi = 3.141592; (PASCAL)
pi: *constant* FLOAT:= 3.141592; (ADA)
var a,b: diasemanal; (PASCAL)
a,b: diasemanal:=martes; (ADA)

En Pascal la utilización de las palabras clave *const* y *var* se hace una sola vez para declarar las constantes y variables. En Ada no se utiliza una palabra clave como introducción a la declaración de datos. La declaración de una constante dinámica en Ada puede realizarse de la siguiente forma:

arriba: *constant* INTEGER: M + 1

Como vemos en el ejemplo anterior, los valores dependen del valor de la variable M cuando esta declaración es procesada.

Un ejemplo de array dinámico podría ser el siguiente:

dinámico: *array* (1..N) *of* FLOAT: = (1 ⇒ 2.0, *others* ⇒ 0.0);

En el ejemplo anterior el tamaño del array depende del valor de N cuando esta declaración es procesada.



EXPRESIONES Y OPERADORES

En Ada existen, como en Pascal, operadores aritméticos, relacionales y booleanos. Veamos una primera diferencia de conexión de estos operadores en Ada y Pascal.

if (k<0) and (j>10) *then*... (PASCAL)
if k<0 *and* j>10 *then*... (ADA)

Como se observa, en Pascal son necesarios los paréntesis porque el operador *and* tiene nivel de prioridad más alto que los operadores relacionales. En Ada los operadores booleanos tienen el nivel de precedencia más bajo.

Tanto en Ada como en Pascal, los operadores relacionales se definen para objetos escalares y para cadenas, es decir, arrays de caracteres.

SENTENCIAS DE CONTROL

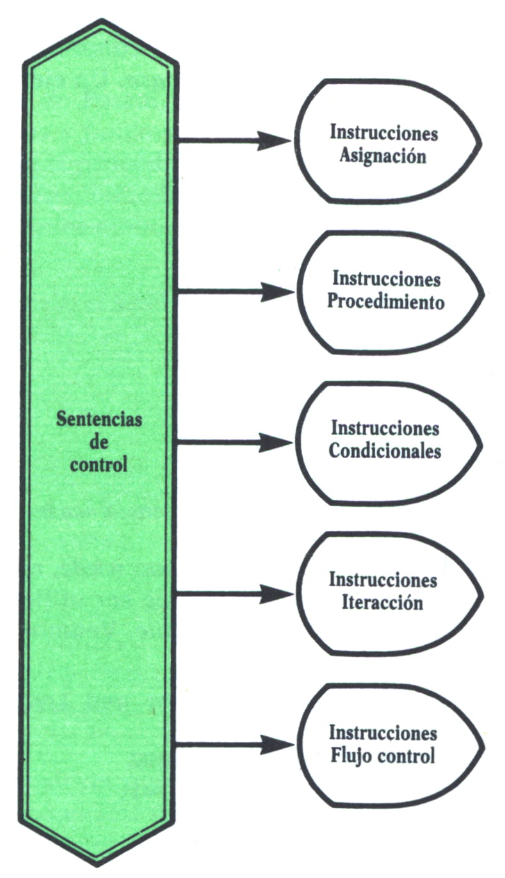


Fig. 4. Sentencias de control.

Las sentencias de control que ambos lenguajes aportan son:

— *Instrucciones de asignación.*
— *Instrucciones de procedimiento.*
— *Instrucciones condicionales.*
— *Instrucciones de iteración.*
— *instrucciones de flujo de control.*

Ada permite asignaciones a partes de un array (llamadas <<slices>>). Veamos un ejemplo de esto último.

type mes *is array* (1..31) *of* INTEGER; (ADA)
meses: mes
meses (1..4):=(7,7,9,5)

Las instrucciones de procedimiento son muy similares en ambos lenguajes. En Ada las instrucciones condicionales son de la forma *if* y *case*. Existen pequeñas diferencias con Pascal, tales como finalizar cada bloque *if* con *end if* y cada sentencia *case*, con *end case*. Un ejemplo de estas sentencias podría ser:

```
if z>0 and f>g
then t:=t+1;
else t=t-1;
end if;
case y is
when rojo => y:=amarillo;
when rosa => y:= naranja;
end case;
```

Programa 1. Ejemplo de sentencia «case».

Las instrucciones de *iteración* en Pascal son *while, repeat* y *for*. Ada tiene una única instrucción de bucle que puede ser utilizada por sí misma, o puede ser prefijada con cláusulas *for* o *while*. Veamos un ejemplo.

```
for m in 1..100 loop B(m):=m; end loop
while j>0 and then S(j)>0 loop
      S(j):= S(j) - 1 ; j:= j+1;
end loop
d:=0; loop d:=d+1; F(d):=d;end loop
```

Programa 2. Ejemplo de iteración.

Ada no posee la instrucción *repeat* del Pascal. En el último ejemplo observamos que no hay condición para que el bucle finalice, por lo que se produciría un error al alcanzar el array sus límites. Este bucle infinito puede ser abortado insertando una instrucción condicional, la cual realiza un chequeo para la finalización en unión con una instrucción *exit, return* o *goto*.

d:=0; *loop* d:=d+1; F(d):=d; *exit when* d= 50;*end loop;*

Este último ejemplo tiene todas las características de una instrucción *repeat*. Ada proporciona una salida incondicional a través de la instrucción *goto*.



SUBPROGRAMAS

En Ada, como en Pascal, los subprogramas pueden estar constituidos por *procedimientos* y *funciones*. La estructura general es similar, una cabecera de función o procedimiento, unas declaraciones y una secuencia de instrucciones. Las llamadas a los subprogramas son similares también. Tan sólo existen entre ambos lenguajes pequeñas diferencias sintácticas.

```
procedure P (a:integer; var b:integer);
     declaraciones locales
begin
                                             (PASCAL)
     secuencia de instrucciones
end;

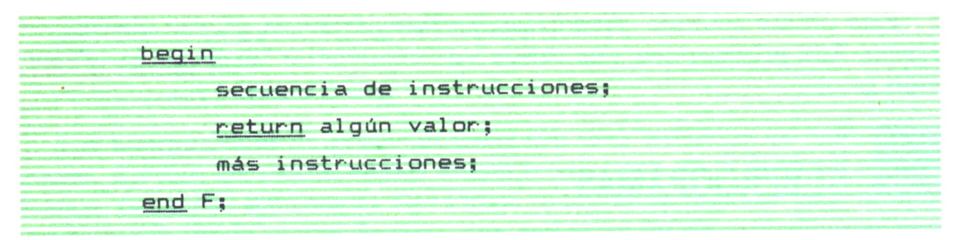
procedure P(a: in INTEGER; b: out INTEGER) is
     declaraciones locales
begin
                                                       (ADA)
     secuencia de instrucciones
end P;
```

Programa 3. Ejemplo de subprogramas.

En Pascal los valores que una función devuelve se asignan a la variable implícita que tiene el mismo nombre que la función. En Ada los valores de las funciones son explícitamente devueltos por medio de la instrucción RETURN. Un ejemplo aclarará esto:

```
     function F(x:integer; var y:integer):integer;
          declaraciones locales
     begin                                           (PASCAL)
          secuencia de instrucciones
     end

     function F (x,y :in INTEGER) return INTEGER is
          declaraciones locales;
```



```
begin
     secuencia de instrucciones;
     return algún valor;
     más instrucciones;
end F;
```

Programa 4. Ejemplo de funciones.

En Ada no se permiten «efectos laterales» sobre los parámetros, pero sí sobre los datos globales. Esto significa que las funciones en Ada pueden tener solamente parámetros *in*.

Ada permite utilizar nombres idénticos para procedimientos y funciones, al contrario de lo que sucede en Pascal, que ambos nombres deben ser únicos.



REGLAS DE ALCANCE

El Pascal tiene las reglas del Algol 60, pero Ada tiene una variedad mucho más amplia de unidades de alcance que Pascal (por ejemplo, bloques, paquetes, tareas, etc.). Todas estas reglas extendidas serán comentadas en capítulos posteriores.



PRINCIPIOS DE LA INGENIERIA DEL SOFTWARE

Los principios de la ingeniería del software están relacionados con el diseño, organización, implementación y mantenimiento de grandes sistemas. El principal interés de la ingeniería del software es dar una solución a la complejidad de estos grandes sistemas.

Las investigaciones de la ingeniería del software demostraron, en la pasada década, que la construcción de programas únicos y monolíticos daban lugar a un producto de difícil utilización y mantenimiento.

Se llegó a la conclusión de que un sistema debía ser dividido en un número de componentes que pudiesen ser tratados de manera independiente.

Para entender las características que deberían incluirse en un lenguaje de programación para que éste sirviese de soporte a la ingeniería del software, sería de utilidad repasar los conceptos que surgieron en la pasada década. Estos conceptos son:

— *Modularidad.*
— *Abstracción.*
— *Concurrencia.*

— *Tratamiento de excepciones.*
— *Interfase con el hardware.*

Expliquemos un poco estos conceptos, ya mencionados en otros capítulos, introduciendo los conceptos de Ada que los soportan.



MODULARIDAD

Como ya hemos comentado, la *modularidad* consiste en dividir un gran sistema en partes o módulos, de manera que podamos hacer un tratamiento de cada una de estas partes de forma independiente. Ada soporta esta necesidad a través de: *unidades de programa* y por *compilación separada*. Junto a las tradicionales construcciones de modularización, tales como subprogramas (procedimientos y funciones), Ada proporciona dos construcciones como son las *tareas* y los *paquetes*.

Un *paquete* (package) está formado por dos partes diferenciadas: la *especificación* (visible) y el *cuerpo* (implementación). La parte visible es accesible al usuario del paquete y puede contener declaraciones de tipos, datos, subprogramas, tareas y paquetes. Sin embargo, la *implementación* es oculta al usuario. Por ejemplo, no hay un camino a través del cual acceder a los datos locales del cuerpo (implementación) de un paquete. Veamos un ejemplo:

```
package P is
   -- tipos , datos, subprogramas, etc. (visible)
end P;
package body P is
        -- datos      locales,subprogramas,      paquetes,
        declaración de tareas
        -- implementación   (visible)   de   subprogramas,
        paquetes, tareas
begin
        -- inicializaciones locales
end P;
```

Programa 5. Ejemplo de «paquete».

En Ada no es necesario que la implementación vaya inmediatamente después de su parte visible. Podemos primero escribir un conjunto de partes visibles para, posteriormente, añadir los cuerpos de implementación

al final del programa. De esta forma un programador puede inicialmente concentrarse sobre la especificación de los componentes del sistema y preocuparse de los detalles de implementación posteriormente.

La posibilidad de poder separar la especificación del cuerpo es una característica de Ada que permite compilar la parte visible de un paquete, independientemente de su cuerpo de implementación.

Las compilaciones separadas junto con la separación de la parte visible de la implementación proporcionan el medio de especificar las interfases («interfaces») de los diferentes módulos de un sistema. Esto es importante en el desarrollo de grandes sistemas con gran número de programadores, porque permite a éstos desarrollar y probar módulos del sistema de una forma independiente.



ABSTRACCION

La abstracción es una herramienta que permite enfocar nuestra atención sobre aspectos diferentes de los objetos en diferentes tiempos. Una forma de abstracción se encuentra en los lenguajes de programación. Esta forma se soporta a través de procedimientos, funciones o subrutinas.

Ada proporciona dos formas de abstracción que raramente se encuentra en otros lenguajes de programación: *abstracción de datos* y *abstracción de tipos*. Distinguiremos una forma «débil» y otra «fuerte» de abstracción de datos: *encapsulación de datos* y abstracción estructural. Veamos un ejemplo.

```
package Apilamiento
    type pila is array (1..300) of FLOAT;
    procedure PUSH (f:in FLOAT;s : in out pila);
    procedure POP (f : out FLOAT; s : in out pila);
end Apilamiento;
```

Programa 6. Ejemplo de abstracción.

La forma más potente de abstracción de datos, abstracción estructural, normalmente incluye encapsulación de datos, pero a la vez oculta la estructura del tipo encapsulado. Por ejemplo:

```
package Apilamiento is
    type pila is private
```

```
        procedure PUSH (f: in FLOAT; s : in out pila);
        procedure POP (f: out FLOAT; s : in out pila);
            ...........
    end Apilamiento
```

Programa 7. Ejemplo de abstracción estructural.

La diferencia entre estos dos ejemplos reside en que en el primero uno puede operar directamente sobre cualquier elemento de una pila, mientras que en el segundo se tiene acceso a la pila solamente a través de las operaciones que proporcionan PUSH y POP.

La abstracción de datos se utiliza a menudo como regla para dividir un sistema en unidades coherentes.

La abstracción de tipos es un requisito crucial para escribir software reutilizable. En lenguajes que no poseen abstracción de tipos el programador debe reescribir la especificación completa y la implementación de un paquete cuando se cambia algún pequeño detalle, tal como el tipo de elemento de la pila en el ejemplo anterior. Por ejemplo, podríamos crear un esquema de programa para pilas que es independiente del tipo de elemento de la pila.

```
generic type elemento is private
package ModuloPila is
    type pila is private
    procedure PUSH (e: in elemento; s: in out pila);
    procedure POP (e: out elemento; s: in out pila);
    ...........
    end ModuloPila;
```

Programa 8. Ejemplo de programa independiente del tipo de elemento.

En este momento podríamos declarar una variedad de módulos de pila, así:

package PilaReal *is new* ModuloPila (FLOAT);
Package PilaEntera *is new* ModuloPila (INTEGER);

Las *unidades genéricas* son especialmente útiles si uno desea definir una estructura y operaciones sobre dicha estructura, independientemente del tipo particular de elementos que puedan ser almacenados en la estruc-

tura. Esta forma de abstracción de tipos es de gran utilidad para el propósito de escribir software reutilizable.



CONCURRENCIA

Hay problemas que sólo pueden ser resueltos utilizando procesos concurrentes, tales como sistemas operativos, reserva de plazas en compañías aéreas, sistemas expertos, etc.

Los sistemas que requieren procesos concurrentes son muy difíciles de escribir correctamente en lenguajes que no soportan concurrencia. Los problemas se presentan en áreas tales como acceso a datos y recursos compartidos.

Ada es uno de los pocos lenguajes de programación que proporciona el soporte adecuado para los procesos concurrentes. La construcción que soporta la concurrencia es la *tarea* (task).

En su estructura, la tarea es similar al paquete (package); está formada por una *parte visible* y un *cuerpo* (implementación). De manera similar al paquete, en la tarea la especificación está completamente separada de la implementación. Esta separación hace de la tarea una construcción muy útil para la modularización. Con respecto a la compilación separada, las tareas solamente pueden ser *subunidades,* no unidades de librería.

Sin embargo, hay dos diferencias básicas entre tareas y paquetes: en contraste con el cuerpo del paquete, el cuerpo de una tarea define una actividad independiente y la interfase de la tarea define un número de puntos de entrada dentro de la tarea. Supongamos la tarea que gestiona una reserva en una compañía aérea.

```
task ListaVuelo is
        entry RESERVA (s:out asiento);
        entry CANCELAR (s: in asiento);
end ListaVuelo;
```

Programa 9. Ejemplo de «tarea».

Otras tareas comunican con la tarea ListaVuelo llamando a esos puntos de entrada de una manera similar a las llamadas a procedimientos.

ListaVuelo.RESERVA(x);
ListaVuelo.CANCELA(y);



TRATAMIENTO DE EXCEPCIONES

En grandes sistemas el tratamiento de errores es de significativa importancia. Cada programa está formado por dos clases de partes: aquellas partes que implementan el algoritmo general y aquellas otras que gestionan los errores (terminación anormal del algoritmo). Los programas en los cuales estas dos partes intervienen son, evidentemente, más complejos que aquellos otros en los cuales estas dos partes están separadas. Ada proporciona un mecanismo de tratamiento de excepciones que hace posible separar estas dos clases de partes de un programa. En Ada, el código de tratamiento de errores siempre es la última parte de un bloque de programa, separado de la parte general.



INTERFACES CON EL HARDWARE

En la mayoría de los lenguajes no se proporciona un medio para la representación de las interfases hardware. Este soporte sólo puede encontrarse fuera del lenguaje, normalmente utilizando codificación en lenguaje ensamblador.

Ada proporciona facilidades tales como representar estructuras lógicas sobre representaciones físicas, datos pueden ser representados sobre direcciones particulares y las entradas pueden ser representadas sobre interrupciones particulares.

ELEMENTOS, TIPOS DE DATOS Y DECLARACIONES DE OBJETOS 5

ELEMENTOS DEL LEXICO





DA tiene una estructura léxica bien definida, la cual forma su base y que permite mejorar la inteligibilidad del lenguaje y su transporte a otras máquinas. El texto de un programa Ada está basado en un conjunto de elementos que comparten una serie de características comunes, tales como:

— Para el conjunto de caracteres el texto de un programa solamente puede incluir caracteres gráficos ASCII.

— El conjunto completo de caracteres (gráficos y caracteres de control) puede ser utilizado a través de tipos predefinidos.

— No se requieren marcas de continuación para seguir con el texto del programa a través de sucesivas líneas.

— Espacios y líneas en blanco se permiten para mejorar la lectura de los programas.

Los elementos del léxico de Ada que permiten la construcción de programas en este lenguaje se dan a continuación:

Delimitadores

Utilizados como separadores y operadores. Dentro de éstos tenemos los simples y compuestos.

Delimitadores simples:

& ' ( )* +, - . / : ; < = >

Delimitadores compuestos:

=> .. ** := /= >= <= << >> <>

Identificadores

Utilizados como nombre de las entidades dentro del texto del programa. Deben comenzar con una letra, si bien los sucesivos caracteres pueden ser letras, dígitos o subrayado.

El subrayado solamente puede aparecer entre dos letras o dígitos o entre una letra y un dígito. Las letras mayúsculas o minúsculas son tratadas de la misma forma. Los identificadores pueden ser tan largos como una línea e incluyen nombres definidos por el usuario y palabras reservadas.

Literales enteros

Representan un literal numérico sin punto decimal. Los valores permitidos son los números positivos y el cero. Pueden representarse en cualquier base de 2 a 16. Deben comenzar con un dígito (el subrayado está permitido entre dígitos).

Literales reales

Representan un literal numérico con punto decimal. Los valores permitidos son los números positivos y el cero. Pueden representarse en cualquier base de 2 a 16. Pueden tener exponente positivo o negativo y deben comenzar con un dígito (el subrayado está permitido entre dígitos).

Caracteres

Representan uno de los 95 caracteres gráficos (ASCII).

Este tipo de datos es un tipo carácter. Se construyen literales de caracteres encerrando un simple carácter entre apóstrofos.

Cadenas

Representan una secuencia de cero o más caracteres encerrados entre comillas. Deben colocarse en un simple y las comillas pueden ser incluidas en la cadena colocando dobles comillas.

Comentarios

No tienen efecto sobre el programa. Comienzan con dos guiones seguidos y el comentario se extiende hasta el final de la línea.

Veamos algunos ejemplos:

Identificadores:

Librería de matemáticas	— el nombre de un paquete
Coseno	— el nombre de una función
Velocidad	— el nombre de una variable

Enteros:

4567	— un valor de 4.567
4 567	— el mismo valor anterior
2500	— un valor de 2.500
25E2	— el mismo valor anterior
2#1111 1111#	— notación en base 2 para el 255
16#FF#	— notación en base 16 para el 255

Reales:

3456.0	— un valor real 3.456,0
3 456.0	— el mismo valor anterior
26.0E2	— un valor de 2.600,0
2#1111 1111.0#	— notación en base 2 para el 255

Caracteres:

'A '	— letra mayúscula A
'a '	— letra minúscula a
' '	— carácter espacio

Cadenas

«Esto es una cadena» «B» — Esto es la cadena B no el carácter B

Cuando todos estos elementos se unen forman las instrucciones específicas, las declaraciones y las unidades de programa. Algunos fallos se producen cuando intentamos utilizar una palabra reservada como identificadores definidos por el usuario. Se recomienda revisar la lista de palabras reservadas para evitar estos errores. También se recomienda utilizar identificadores que sean coherentes con el tipo de aplicación que se está desarrollando, se debe huir de identificadores complejos y rebuscados. De igual manera, la indentación y el espaciamiento ayudan a una mejor lectura del programa.

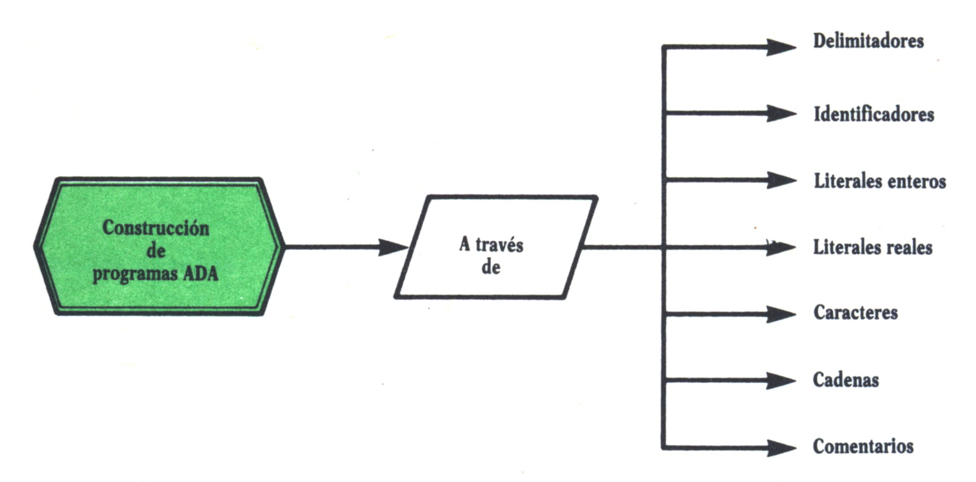


Fig. 1. Construcciones de programas en ADA.

Tipos de datos

Ada posee una amplia variedad de tipos de datos. El lenguaje permite al usuario definir sus propios tipos de datos utilizando la sintaxis que mostraremos al final del libro. Como con cualquier otro lenguaje de programación, Ada proporciona al usuario un número de tipos definidos por el lenguaje y que pueden utilizarse en el desarrollo normal de un programa. A continuación damos una visión general de los tipos de datos soportados por Ada

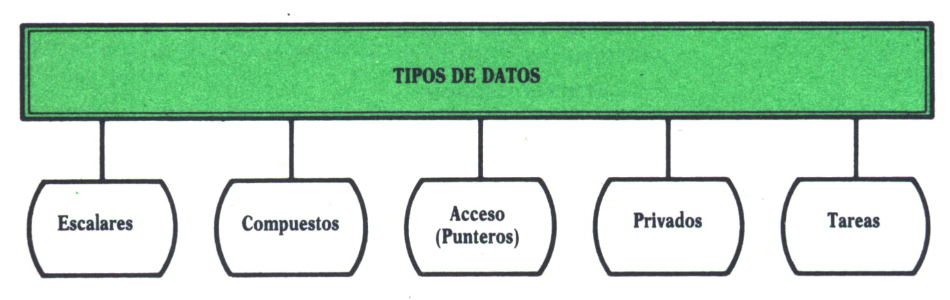


Fig. 2. Tipos de datos en ADA.

Tipos de datos

Caracterizan un conjunto de valores y el conjunto de operaciones permitidas sobre ellos.

Tipos escalares

Tienen un orden predefinido con un margen de valores contiguos. Dentro de los tipos escalares tenemos los *discretos, enumerados, carácter, enteros, reales.*

Tipos discretos

Representan valores dentro de un número finito de posiciones discretas.

Tipos enumerados

Representan un conjunto ordenado de diferentes valores especificados por su lista.

Tipos carácter

Representan tipos enumerados en los cuales uno de sus valores permitidos es un carácter.

Tipos enteros (tipo numérico)

Representan un conjunto consecutivo de enteros sobre un limitado margen de valores.

Tipos reales

Representan una aproximación a valores numéricos continuos del mundo real dentro de una precisión especificada. Entre ellos tenemos los tipos en *coma flotante* y en *coma fija.* Los primeros presentan errores de precisión que varían con la magnitud del valor representado. Los segundos tienen errores de precisión constante para todos los valores representados.

Tipos compuestos

Poseen una estructura formada por otro tipo de datos cuyos valores se determinan por los valores de sus componentes. Entre éstos tenemos los *tipo array* y *tipo registro.*

Tipo array

Representan un conjunto indexado de componentes similares.

Tipo registro

Representan un conjunto de componentes de tipos diferentes.

Tipo acceso (punteros)

Poseen valores que designan objetos de otros tipos.

Tipos privados

Poseen valores cuya representación fundamental es transparente (oculta) al usuario del tipo.

Tipos tareas

Poseen valores que designan tareas.

Veamos algunos ejemplos de estos tipos de datos:

```
-- definiciones de tipo
(BLANCO,ROJO,AZUL,GRIS)                  -- enumerados
range 1..70                    -- entero
array (1..15)                  -- array
-- definiciones completas de tipo
type COLOR is (BLANCO,ROJO,AMARILLO,GRIS);
type fila is range  1..70;
type TABLA is array (1..15) of INTEGER;
-- Ejemplos de tipos enteros
     type NUMERO is range 1..3_000;
     type LINEA is range 1..TAMAÑOMAXIMO;
     type INDICE is range IND_MIN..MAX*K
-- Ejemplos de tipos enumerados
     type DIA is (LUNES,MARTES,MIERCOLES);
```

```
    type NIVEL is (BAJO,MEDIO,ALTO);
    type LUZ is (ROJA,AMBAR,VERDE);
-- Ejemplos de tipos en coma flotante

    type COEFICIENTE is digits 1Ø range -1.Ø .. 1.Ø;
    type REAL is digits 8;
-- Ejemplos de tipos en coma fija
    type VOLTIOS is delta Ø.125 range Ø.Ø .. 22Ø.Ø;
    type VELOCIDAD is delta Ø.2 range -13.Ø .. 15.Ø;
-- Ejemplos de tipos array
    type VECTOR is array (1..2Ø) of INTEGER;
    type LINEA is array (1..MAXIMO) of CHARACTER;
    type MATRIZ is array (DIA range LUNES..VIERNES, INTEGER
    range -1Ø..1Ø) of REAL;
-- ejemplo de tipos registro
    type FECHA is
         record
              DIA : INTEGER range 1..31;
              MES : NOMBRE MES;
              AÑO : INTEGER range Ø..2ØØØ;
         end record;
```

Programa 1. Ejemplos de distintos tipos de datos.

Los registros con *discriminantes* permiten modificar la estructura de los objetos del registro.

Un ejemplo de esto puede ser:

type BUFFER (TAMAÑO : TAMAÑO BUFFER:=100) is
record
POS: TAMAÑO BUFFER:=0;
VALOR: CADENA (1..TAMAÑO);
end record;

Los tipos de registro pueden ser componentes de otros tipos, incluyendo tipos registro o tipos array.

```
-- Ejemplos de tipos acceso (PUNTEROS)
    type ARMAZON is access MATRIZ;
    En el ejemplo anterior objetos de tipo ARMAZON, apuntan
a objetos de tipo MATRIZ.
```

```
-- Ejemplos de tipo privado
      type CLAVE is private
Un  tipo privado solamente puede ser declarado en un  número
limitado de sitios,  normalmente  en la especificación de un
paquete en el cual exista una parte privada.
      package CLAVE_NUMERO is
            type CLAVE is private
            CLAVE_NULA : constant CLAVE;
            procedure OBTENER_CLAVE (K:out CLAVE);
            private
                  -- declaraciones completas de tipo
            type CLAVE is new NATURAL;
            CLAVE_NULA : constant CLAVE:=Ø;
      end CLAVE_NUMERO;
```

Programa 2. Ejemplo de tipos acceso y privado.

Un tipo privado tiene un conjunto restringido de operaciones permitidas. Podemos además restringir estas operaciones haciendo el tipo privado *limitado*.

```
      type NOMBRE is limited  private;

-- Ejemplos de tipos tarea
      task type RECURSOS is
            entry TAMAÑO; -- las  entradas definen los  puntos
                              -- de interacción entre tareas
            entry LIBERAR;
      end RECURSOS;
      task type TECLADO is
            entry LEER (C:out CARACTER);
                  -- parámetros  pueden  ser pasados  en  estos
                  -- puntos de entrada entre tareas.
            entry ESCRIBIR (C: in CARACTER);
      end TECLADO;
```

Programa 3. Ejemplo de tipo privado limitado.

Una declaración de *subtipo* añade una restricción a un tipo de datos, sin introducir un nuevo tipo de datos diferente. Los objetos de un *subtipo* pueden ser mezclados con objetos de su tipo base u otro subtipo de su tipo base. Tales mezclas no requieren conversiones de tipo. La declaración de un subtipo es de la forma:

subtype Nombre del subtipo *is* Nombre del tipo base

Los *tipos derivados* crean un tipo distinto de dato. La forma general de este tipo es:

type derived Nombre de tipo *is new* Nombre del tipo base

Veamos algunos ejemplos:

subtype PEQUEÑO ENTERO *is* INTEGER range -10..10;
subtype ARCO IRIS *is* COLOR *range* ROJO..AZUL;
subtype CUADRADO *is* MATRIZ (1..15,1..15);

En este último ejemplo la restricción sobre el tipo array aplica solamente a los márgenes de los índices para tipos no restringidos.



DECLARACION DE OBJETOS

Ada utiliza el concepto de *objeto* para referirse a las variables y constantes existentes dentro de un programa. Los objetos pueden ser vistos como grandes «cajas» las cuales guardan datos que serán manipulados por un programa. Estas «cajas» mantienen más información que el simple valor del objeto.

Los objetos poseen las siguientes propiedades:

— *Un identificador,* que es el nombre que utilizamos para referirnos al objeto.

— *Un tipo de datos,* que es el tipo que restringe la forma y los valores permitidos por el objeto.

— *Un conjunto de operaciones,* que son las operaciones permitidas sobre el objeto. Estas operaciones pueden ser operaciones predefinidas y las proporcionadas por el usuario.

— *Un valor inicial,* es decir, el valor que un objeto asumirá cuando éste se defina.

— *Un conjunto de atributos,* que puede contemplarse como un conjunto de funciones que permiten al programa tener acceso a las propiedades de un objeto o a su tipo de datos.

Algunos ejemplos sobre declaración de objetos pueden ser los siguientes:

— Los objetos pueden ser variables o constantes.

— Declaración de variables:

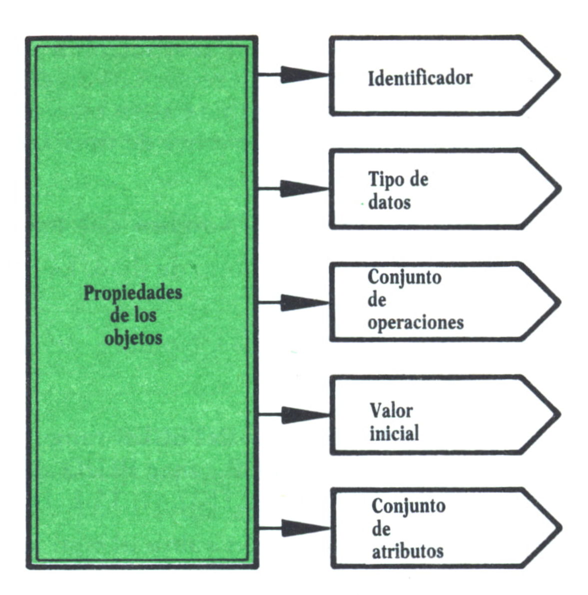


Fig. 3. Propiedades de los objetos.

SUMAR, RESTAR : INTEGER;
LONGITUD : INTEGER *range* 0..100 :=0;
COLOR : BOOLEAN :=FALSE;

— Las constantes son asignadas a valores en sus declaraciones.

ANCHO : *constant* INTEGER:= 88;
ANCHO PEQUEÑO : *constant* INTEGER:=ANCHO/15;

— Múltiples objetos pueden ser declarados en una única declaración.

AZUL,ROJO: NOMBRE COLOR:= *new* COLOR (TINTE => Fuerte);
sería equivalente a:

AZUL: NOMBRE COLOR:= *new* COLOR (TINTE => Fuerte);
ROJO: NOMBRE COLOR:= *new* COLOR (TINTE => Fuerte);

— En la declaración de objetos array se debe incluir un índice que restrinja el tipo de array.

CUADRADO : MATRIZ (1..10,1..15);

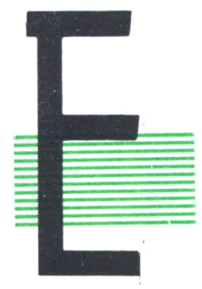
— Pueden inicializarse valores de un array.

X: TABLA := (3,4,5,6,0) -X(1)=3,X(5)=0

— Variables registro.

MAÑANA,PASADOMAÑANA:FECHA;
GRANDE : BUFFER(100); -restringido a 100 caracteres

UNIDADES DE PROGRAMA Y DECLARACIONES 6



EN este capítulo estudiaremos las piezas básicas que constituyen un programa Ada, su estructura general y la forma de declarar cada una de estas piezas.



UNIDADES DE PROGRAMA: SU ESTRUCTURA

En el lenguaje Ada, todas las unidades de programa tienen la misma estructura lógica. Sin embargo, pueden existir algunas variaciones en la realización física de algunas unidades de programa individuales. Cada *unidad de programa* consiste en:



ESPECIFICACION

Esta parte de la unidad de programa describe a las personas que harán uso de la unidad de programa «que hace la unidad de programa».



CUERPO

Esta parte detalla cómo la unidad de programa llevará a cabo la implementación de operaciones, algoritmos o estructuras recogidas en la parte de especificación.

El motivo de separar la *especificación* del *cuerpo* es incrementar la fiabilidad y mantenibilidad de los programas. La fiabilidad de un programa aumenta, ya que una causa muy común de error, «errores de interfase», pueden fácilmente ser detectados en tiempo de compilación y no es nece-

sario esperar a detectarlos cuando se producen las pruebas del sistema. La mantenibilidad de un programa se incrementa, ya que si se producen cambios en ciertos detalles de la implementación, éstos se restringen al cuerpo de la unidad de programa sin que esto afecte al usuario de la unidad de programa.

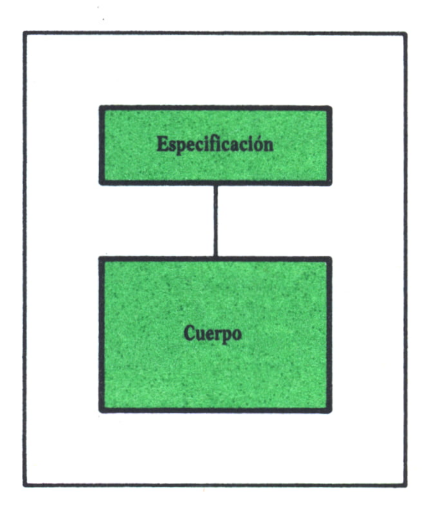


Fig. 1. Estructura general de una unidad de programa.

Como *unidades de programa* tenemos las siguientes:

— *Subprogramas.*
— *Procedimientos.*
— *Funciones.*
— *Paquetes.*
— *Tareas.*



GENERICOS

A continuación damos una serie de características y uso de las unidades de programa.

* Los *subprogramas* son las estructuras básicas ejecutables de un programa. Pueden tener la *especificación* y el *cuerpo* separados. La especificación constituye la «interface», y el cuerpo, la implementación. Los usos de esta unidad de programa se extienden a los programas principales, definición de algoritmos y operadores.

* Los *procedimientos* definen una secuencia de acciones y se invocan a través de una llamada al procedimiento.

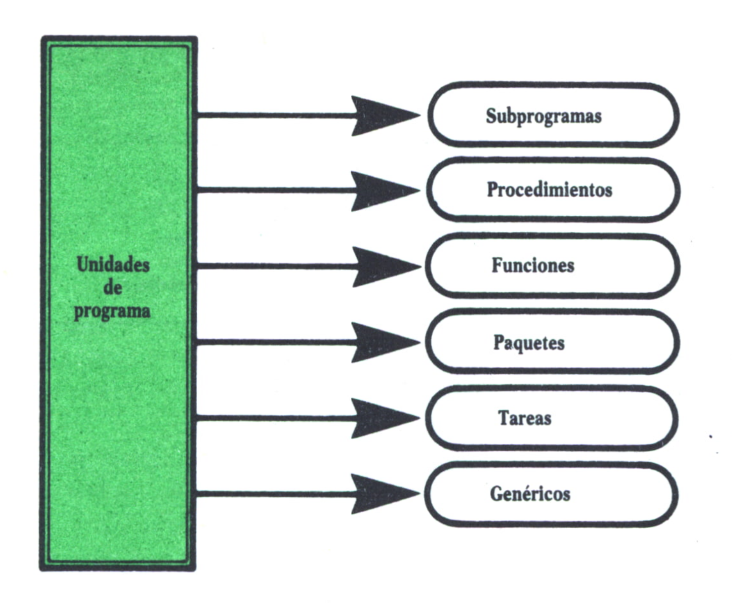


Fig. 2. Unidades de programa en ADA.

* Las *funciones* definen un cálculo retornando un valor. Se invocan desde el interior de una expresión.

* Los *paquetes* son las unidades estructuradas básicas de Ada. El cuerpo y la especificación están separados. La parte visible o especificación es accesible al usuario del paquete y puede contener declaraciones de tipos, constantes, subprogramas, tareas, etc. La parte de implementación es oculta al usuario. Los usos del paquete van desde establecer conexión entre grupos de unidades de programa a los tipos abstractos de datos.

* Las *tareas* se ejecutan concurrentemente con otras tareas. Se utilizan en Ada para dar soporte a los procesos concurrentes. Poseen mecanismos para la sincronización y la transmisión de datos. Sus usos se extienden a la *concurrencia* de acciones y procesos, el acceso controlado a recursos compartidos, la gestión de interrupciones y a los buffers, colas y pilas FIFO.

* Los *genéricos* son esquemas parametrizados de otras unidades de programa. En Ada un esquema de programa puede escribirse como una unidad de programa genérica formada de una cláusula genérica que precede a un subprograma, un paquete o una tarea. La cláusula genérica especifica los parámetros genéricos, que pueden ser datos (expresiones, variables o constantes), parámetros tipo, parámetros procedimiento o parámetros funciones. Los principales usos de los *genéricos* son la reutilización de componentes y las librerías.

Veamos algunos ejemplos de las unidades de programa mencionadas hasta el momento:

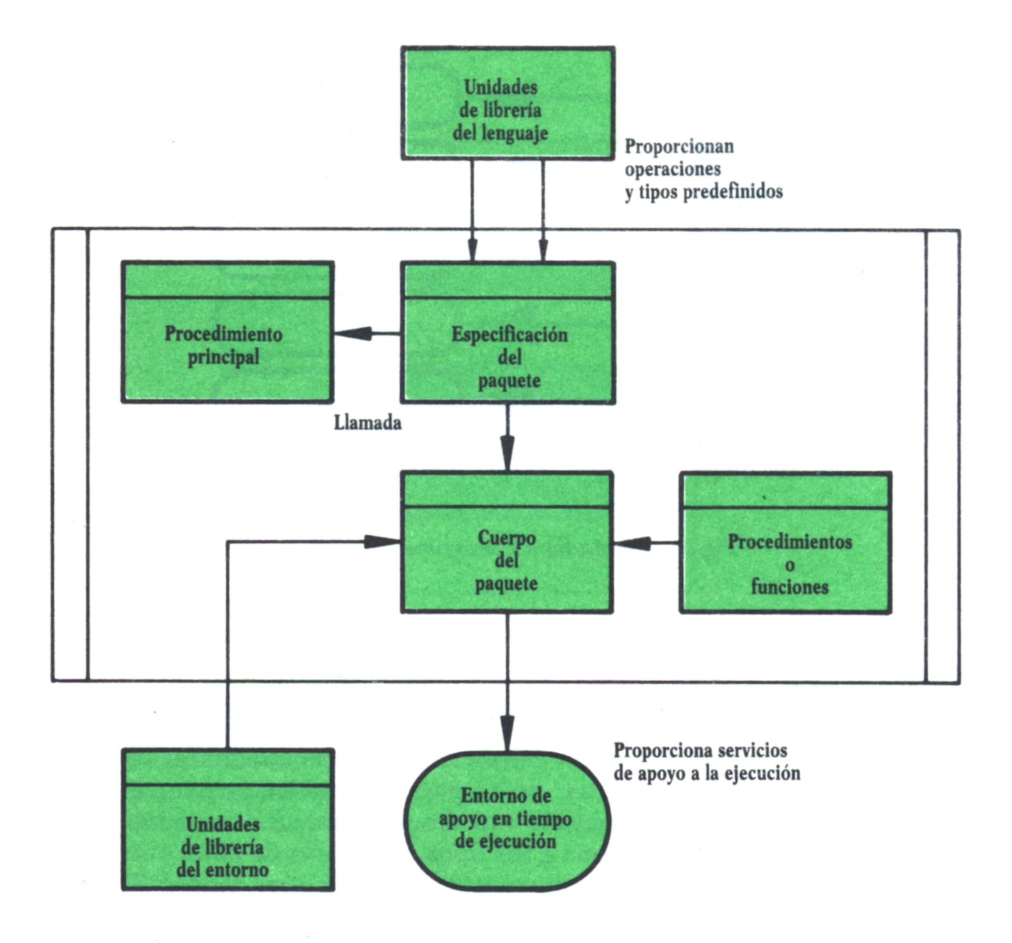


Fig. 3. Esquema de un programa ADA.

```
-- Procedimientos
    ESPECIFICACION
        procedure PUSH (A: in ELEMENTO; B: in out PILA);
    CUERPO
        procedure PUSH (A: in ELEMENTO; B: in out PILA) is
         begin              -- comienza parte ejecutable
          if B.INDICE = B.TAMAÑO then
         raise DESBORDAMIENTO;
```

```
            else
            B.INDICE := B.INDICE + 1;
            B.ESPACIO(B.INDICE):= A;
            end if;
            exception
                when CONSTRAINT_ERROR => raise;
        end PUSH;

  -- Funciones
   ESPECIFICACION
  function  PRODUCTO (DERECHO,IZQUIERDO :  VECTOR) return REAL;
   CUERPO
    function PRODUCTO (DERECHO,IZQUIERDO:VECTOR) return REAL is
       SUMA:REAL:=Ø.Ø
       for J in IZQUIERDO'RANGE loop
       SUMA:=SUMA + IZQUIERDO(J) * DERECHO(J);
       end loop;
       return SUMA;
  end PRODUCTO;

  -- Paquetes
       ESPECIFICACION DEL PAQUETE
       package GESTION_VECTOR is
        subtype indice is INTEGER range Ø..INTEGER'LAST;
        type vector is array (indice range <> of FLOAT;
        Dimension  :exception;
        function "+" (u,v: in vector) return vector;
        function "-" (u,v: in vector) return vector;
       function "*" (u,v: in vector) return vector;
        function "/" (u,v: in vector) return vector;
        function "*" (x: in FLOAT; v: in vector) return vector;
        function PROD (u,v: in vector) return FLOAT;
      end GESTION_VECTOR;

    CUERPO DEL PAQUETE
    package body GESTION_VECTOR is
      function "+" (u,v: in vector) return vector is
         dim: constant INTEGER:= u'LENGTH;
         w : vector (1..dim);
      begin
         if v'LENGTH /= dim then raise Dimension; end if;
```

```
        for i in 1..dim loop w(i):=u(i)+v(i); end loop;
        return w;
    end "+";
function "*" (x:in FLOAT; v: in vector) return vector is
     dim: constant INTEGER:=v'LENGTH;
     w:vector(1..dim);
    begin
        for i in 1..dim loop w(i):=x * v(i); end loop;
         return w;
     end "*";
     function PROD (u,v: in vector) return FLOAT is
         dim : constant INTEGER:= u'LENGTH;
         sum: FLOAT := 0.0;
     begin
      if v'LENGTH /= dim then raise Dimension ; end if;
     for i in 1..dim loop sum:=sum + u(i) * v(i);end loop;
     return sum;
   end PROD;
end GESTION_VECTOR;

     -- Tareas
     ESPECIFICACION DE LA TAREA
     task type Correo is
         entry DEPOSITAR (msg : in mensaje);
         entry SACAR (msg : in mensaje);
     end Correo;
     CUERPO DE LA TAREA
     task body Correo is
         capacidad: constant :=24;
         subtype indice is INTEGER range 0..(capacidad - 1);
         sup,inf : indice :=0;
         numemen : INTEGER range 0..capacidad :=0;
         caja : array (indice) of mensaje;
     begin
       loop
         select

             when numemen < capacidad =>
              accept DEPOSITAR (msg : in mensaje) do
                  caja(sup):=msg;
                  sup:=(sup + 1) mod capacidad;
```

```
                numemen := numemen + 1;
            end DEPOSITAR;
            or
                when numemen > Ø =>
            accept SACAR  (msg : out mensaje) do
                msg := caja(inf);
                inf := (inf + 1) mod capacidad;
                numemen := numemen - 1 ;
            end SACAR
         end select;
       end loop;
   end Correo;
```

En este ejemplo suponemos que el tipo "mensaje" está definido.

```
   -- Genricos
   ESPECIFICACION DE GENERICOS
       generic
       package PARES is
           type par is private;
            function ENSAMBLAR (x,y: in FLOAT) return par;
           function "+"  (u,v : in par) return par;
       private
           type par is record der,izq : FLOAT; end record
        end PARES;

    CUERPO DEL GENERICO
        package body PARES is     -- no visible al usuario
        generic with function "&"(x,y: in FLOAT) return FLOAT;
        function OPERAPAR(u,v: in par) return par;
        function ENSAMBLAR (x,y: in FLOAT) return par is
        begin return (x,y); end ENSAMBLAR;
        function "+" (u,v: in par) return par is
        new OPERAPAR ("+");
        -- Implementación de OPERAPAR
        function OPERAPAR (u,v : in par) return par is
    begin return (u.izq & v.izq, u.der & v.der);end OPERAPAR;
  end PARES;
   -- Procedimientos
        ESPECIFICACION
```

```
   procedure PUSH (A: in ELEMENTO; B: in out PILA);
CUERPO
   procedure PUSH (A: in ELEMENTO; B: in out PILA) is
    begin           -- comienza parte ejecutable
     if B.INDICE = B.TAMAÑO then
     raise DESBORDAMIENTO;
     else
     B.INDICE := B.INDICE + 1;
     B.ESPACIO(B.INDICE):= A;
     end if;
     exception
          when CONSTRAINT_ERROR => raise;
end PUSH;
-- Funciones
 ESPECIFICACION
function  PRODUCTO (DERECHO,IZQUIERDO :  VECTOR) return REAL;
 CUERPO
  function PRODUCTO (DERECHO,IZQUIERDO:VECTOR) return REAL is
     SUMA:REAL:=Ø.Ø
     for J in IZQUIERDO'RANGE loop
     SUMA:=SUMA + IZQUIERDO(J) * DERECHO(J);
     end loop;
     return SUMA;
end PRODUCTO;

-- Paquetes
  ESPECIFICACION DEL PAQUETE
  package GESTION_VECTOR is
   subtype indice is INTEGER range Ø..INTEGER'LAST;
   type vector is array (indice range <> of FLOAT;
   Dimension  :exception;
   function "+" (u,v: in vector) return vector;
   function "-" (u,v: in vector) return vector;
  function "*" (u,v: in vector) return vector;
   function "/" (u,v: in vector) return vector;
   function "*" (x: in FLOAT; v: in vector) return vector;
   function PROD (u,v: in vector) return FLOAT;
 end GESTION_VECTOR;

CUERPO DEL PAQUETE
package body GESTION_VECTOR is
```

```
      function "+" (u,v: in vector) return vector is
          dim: constant INTEGER:= u'LENGTH;
          w : vector (1..dim);
     begin
          if v'LENGTH /= dim then raise Dimension; end if;
          for i in 1..dim loop w(i):=u(i)+v(i); end loop;
          return w;
    end "+";
function "*" (x:in FLOAT; v: in vector) return vector is
      dim: constant INTEGER:=v'LENGTH;
      w:vector(1..dim);
    begin
          for i in 1..dim loop w(i):=x * v(i); end loop;
          return w;
    end "*";
    function PROD (u,v: in vector) return FLOAT is
          dim : constant INTEGER:= u'LENGTH;
          sum: FLOAT := Ø.Ø;
    begin
     if v'LENGTH /= dim then raise Dimension ; end if;
    for i in 1..dim loop sum:=sum + u(i) * v(i);end loop;
    return sum;
  end PROD;
end GESTION_VECTOR;

    -- Tareas
    ESPECIFICACION DE LA TAREA
    task type Correo is
      entry DEPOSITAR (msg : in mensaje);
      entry SACAR (msg : in mensaje);
end Correo;
CUERPO DE LA TAREA
task body Correo is
      capacidad: constant :=24;
      subtype indice is INTEGER range Ø..(capacidad - 1);
      sup,inf : indice :=Ø;
      numemen : INTEGER range Ø..capacidad :=Ø;
      caja : array (indice) of mensaje;
begin
   loop
```

```
    select
        when numemen < capacidad =>
        accept DEPOSITAR (msg : in mensaje) do
            caja(sup):=msg;
            sup:=(sup + 1) mod capacidad;
            numemen := numemen + 1;
        end DEPOSITAR;
        or
            when numemen > Ø =>
        accept SACAR  (msg : out mensaje) do
            msg := caja(inf);
            inf := (inf + 1) mod capacidad;
            numemen := numemen - 1 ;
        end SACAR
      end select;
    end loop;
  end Correo;
```

En este ejemplo suponemos que el tipo "mensaje" está definido.

```
  -- Genricos
  ESPECIFICACION DE GENERICOS
      generic
      package PARES is
          type par is private;
           function ENSAMBLAR (x,y: in FLOAT) return par;
          function "+"  (u,v : in par) return par;
      private
          type par is record der,izq : FLOAT; end record
      end PARES;

  CUERPO DEL GENERICO
      package body PARES is     -- no visible al usuario
      generic with function "&"(x,y: in FLOAT) return FLOAT;
      function OPERAPAR(u,v: in par) return par;
      function ENSAMBLAR (x,y: in FLOAT) return par is
      begin return (x,y); end ENSAMBLAR;
      function "+" (u,v: in par) return par is
      new OPERAPAR ("+");
      -- Implementación de OPERAPAR
```

```
        function OPERAPAR (u,v : in par) return par is
   begin return (u.izq & v.izq, u.der & v.der);end OPERAPAR;
  end PARES;
```

Programa 1. Ejemplo de unidades de programas.



DECLARACION DE LAS UNIDADES DE PROGRAMA

En Ada, el trabajo asociado con un programa se desarrolla por los distintos tipos de unidades de programa contemplados en el lenguaje. Para poder utilizar una unidad de programa, la unidad deberá ser declarada de forma consistente con las reglas del lenguaje. Las declaraciones para unidades de programa cumplen con los siguientes propósitos:

— Asociar un nombre con la unidad.
— Especificar la «interface» a los servicios que proporciona la unidad de programa.
— De manera indirecta, especificar los servicios que proporciona la unidad de programa. Esta especificación es indirecta, ya que los servicios son solamente identificables por el nombre que se le da a la unidad de programa y los parámetros que se especifican.

La sintaxis y características de las diferentes formas de unidades de programa se dan en los siguientes ejemplos.



DECLARACION DE SUBPROGRAMAS

Los procedimientos pueden tener cualquier número de parámetros formales. Podemos restringir cómo se van a usar los parámetros en el interior del procedimiento, indicando el modo del parámetro.

```
procedure ARBOL;
procedure AMPLIAR (Y : in out INTEGER)
-- Podemos  leer  y  escribir  Y  en  el  cuerpo  del
procedimiento.
procedure CAMINAR (NUMERO : in INTEGER);
-- Sólo  podemos  leer  NUMERO  en  el  cuerpo  del
procedimiento.
```

Programa 2. Ejemplo de procedimiento.

Los parámetros de las *funciones* son siempre de modo *in*, de manera que los parámetros formales sólo pueden leerse en el interior del cuerpo de la función.

function AZAR *return* ALEATORIO;
function MINIMO (Y: ENLAZAR) *return* CELDA;



DECLARACION DE PAQUETES

Las especificaciones del paquete definen la interfase con el usuario del paquete. En general, un paquete puede verse como una unidad que proporciona un conjunto de servicios a los usuarios de dicho paquete. Los paquetes pueden utilizarse para dar definiciones de tipos abstractos de datos (tad).

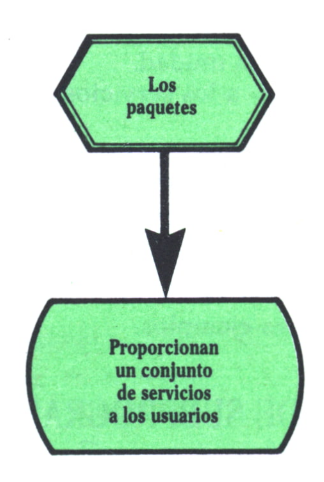


Fig. 4. Paquetes.

```
package HORARIO is  -- se dan tipos y objetos
type DIAS is (LUNES,MARTES,MIERCOLES);
type CONTROL is array (DIAS) of INTEGER;
end HORARIO;

package NUMEROS is
     type RACIONAL is
          record
          NUMERADOR :INTEGER;
```

```
        DENOMINADOR : POSITIVE;
        end record;
function "+" (A,B : RACIONAL) return RACIONAL;
function "/" (A,B : INTEGER) return RACIONAL;
end NUMEROS;
```

Programa 3. Ejemplo de paquetes.

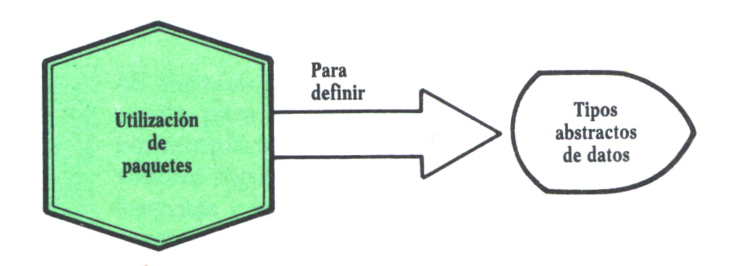


Fig. 5. Utilización de paquetes.



DECLARACION DE TAREAS

Las especificaciones de tareas definen la interfase entre una tarea o tipo de tarea y otras tareas. La interfase ve a las tareas del usuario como un conjunto de procedimientos que se denominan *entradas* («entries»). Una sencilla declaración de tarea es equivalente a declarar un tipo de tarea (sin nombre).

Veamos cómo se especifica una simple tarea:

task RECOLECTAR *is*
entry RECOGER (v: *out* ELEMEN);
entry SEMBRAR (u: *in* ELEMEN);
end RECOLECTAR;

La declaración de tareas como objetos tarea, permiten incluirlas en una estructura de datos.



PRAGMAS Y ESPECIFICACIONES DE TAMAÑO

Las «pragmas» son directivas de compilador que se utilizan para dar información a éste y que aparecen donde existen declaraciones, pero no son

declaraciones como tales. Al final del libro se da una relación de las pragmas contempladas en Ada, así como sus características.



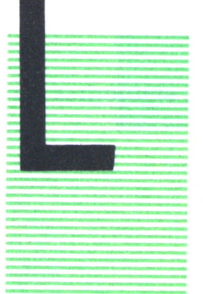
Fig. 6. Pragmas.

EXPRESIONES, OPERADORES E INSTRUCCIONES EN LENGUAJE ADA

7

FORMAS Y USO DE EXPRESIONES EN ADA





LAS expresiones en Ada proporcionan el mecanismo adecuado para el cálculo de un valor. La expresión puede contemplarse como un cálculo que, cuando se ejecute, devolverá algún tipo de dato condicionado por las reglas del lenguaje y la forma de la expresión en sí misma. Las expresiones en Ada se utilizan para una gran cantidad de cosas y entre las cuales citaremos: condiciones para instrucciones de control, parámetros actuales pasados a unidades de programa, etc.

Los nombres son utilizados dentro de una expresión para referirse al valor de un objeto o el valor devuelto por una función. Ejemplos de nombre pueden ser:

CONTADOR	— nombre sencillo de una variable escalar.
PI	— nombre de un número.
DIA.MES	— el componente de un registro.
COCHE.MATRICULA.PROPIETARIO	— un componente de un registro de un componente de un registro.
COLOR'FIRST	— el valor mínimo del tipo enumerado COLOR
FECHA'SIZE	— número de bits para registros de tipo FECHA.
MENSAJE'ADDRESS	— dirección del registro tipo MENSAJE.

Dentro de las expresiones nos encontramos con las llamadas expresiones primarias, tales como:

410.0, que es un literal real
DIV, que es una variable

COSENO (X), que es una llamada a función

De igual forma podemos encontrarnos con distintas formas de expresiones, tales como:

CONTAR *in* ENTERO, que es una relación
A**(B**C), que es una expresión
(ROJO *and* AZUL) *or* BLANCO, que es una expresión

También pueden aparecer en Ada operadores y evaluación de expresiones, tales como:

abs (2 + C) + T
y < 8.0 *and* x > 56.0 es lo mismo que poner
(y < 8.0) *and* (x > 56.0)
A**(–2) en este caso el paréntesis es necesario
X/B * M idéntico a (X/B) * M

Los operadores relacionales y lógicos pueden tomar las siguientes formas:

FRIO *or* CALOR
ESTRUCTURA (1..12) *and* ESTRUCTURA (1..12)
"BB" < "C" *and* "C" < "D" es verdadero
M *not* in 0..20 se prueba si M está dentro de rango
NUMERO_1 = NUMERO_2 verdadero si ambos comparten el mismo número.

El control de rupturas puede expresarse así:

MES.DIA /= null and then MES.DIA.HORA > 30

En este ejemplo la parte de la expresión después de *and then* no se evalúa si la primera parte se evalúa como falsa.

A = 0 *or else* P(A) = VALOR_SUPERIOR

En este ejemplo la parte de la expresión posterior al or else no se evalúa si la primera parte se evalúa como cierta.
Los operadores de adición binaria pueden tomar la forma:

"X" & "Y" da como resultado "XY"

Los asignadores se expresan de la forma:

new TABLA ' (0,*null*,*null*) inicializado explícitamente
new NOMINA no está inicializado

Las expresiones utilizadas para inicializar objetos son de la forma:

PEDRO: NOMBRE_VARON:= *new* NOMBRE (SEXO ⇒ HEMBRA);
FRASE: *constant* CADENA := "HOLA ME LLAMO ANA";



INSTRUCCIONES

Las instrucciones en Ada se utilizan para desarrollar acciones básicas, acciones en tiempo real y para la implementación de una lógica de control.

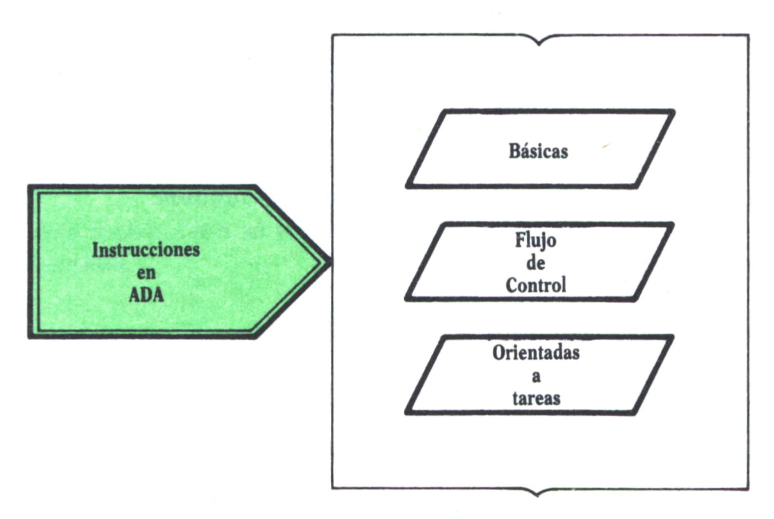


Fig. 1. Instrucciones en ADA.

Estudiaremos, a través de ejemplos, las distintas instrucciones utilizadas en Ada.



INSTRUCCIONES BASICAS

Entre éstas tenemos las instrucciones de *asignación y null.* La primera proporciona un nuevo valor a una variable

nombre de variable := expresión;

La instrucción *null* indica la ausencia de acciones.

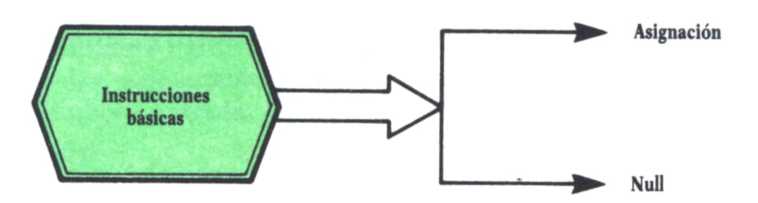


Fig. 2. Instrucciones básicas



INSTRUCCIONES PARA EL FLUJO DE CONTROL

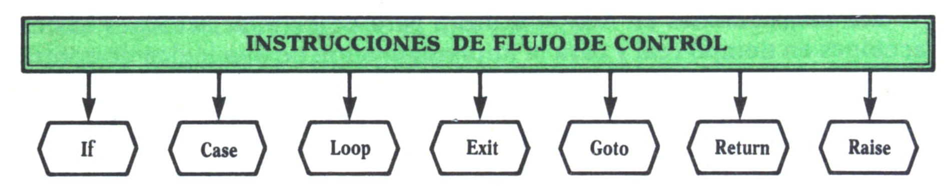


Fig. 3. Instrucciones de flujo de control.

Entre las instrucciones de flujo de control tenemos:

* *IF*

if condición
then
conjunto de sentencias
else
conjunto de sentencias
end if;

* *CASE*

Selecciona una secuencia de instrucciones de varias alternativas que se excluyen mutuamente y basadas en el valor de una expresión discreta.

case expresión is
when opción ⇒ sentencias
when opción ⇒ sentencias
....
when opción ⇒ sentencias
when others ⇒ null;
end case;

«Opción» puede ser un simple valor o un rango de valores de tipo discreto o la palabra *others*.

* *LOOP*

Esta instrucción controla la ejecución reiterada de un grupo de sentencias. La ejecución puede repetirse cero o un número determinado de veces.

loop
sentencias
end loop;
for parámetro del bucle in rango
    loop

sentencias
end loop;
while condición
loop
sentencias
end loop;

* *EXIT*

Solo tiene existencia dentro de un bucle *loop*. Con esta instrucción el control se transfiere a la parte final del bucle.

exit when condición;

* *GOTO*

Con esta instrucción se produce un salto incondicional a una instrucción etiquetada.

goto etiqueta;

* *RETURN*

Esta instrucción nos indica el punto de terminación de un subprograma.

return —en un procedimiento
return expresión —en una función

* *RAISE (Excepción)*

Esta instrucción da lugar a una «excepción» y pasa control al manipulador de excepciones.

raise nombre de la excepción;



INSTRUCCIONES ORIENTADAS A TAREAS

* *ACCEPT*

Identifica la porción de programa que se ejecutará durante un «rendezvous» (procesos paralelos)

accept nombre_entrada *do*
sentencias —ejecutadas durante rendezvous
end nombre_entrada;

* *ABORT*

Provoca la terminación anormal de una o más tareas y de esta manera se evita el procesamiento paralelo con esas tareas.

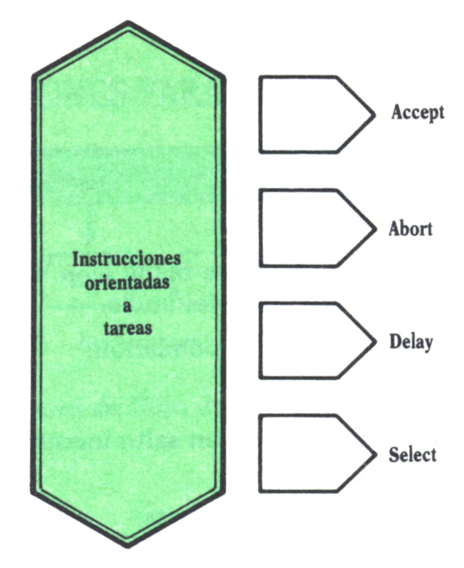


Fig. 4. Instrucciones orientadas a tareas.

abort nombre_tarea;

* *DELAY*

Suspende la ejecución de la tarea, como mínimo, el tiempo especificado en ella.

delay expresión;

* *SELECT*

Esta instrucción controla, de todos los procesos paralelos alternativos, cuál de éstos será ejecutado.

select
instrucción de llamada ("entry call")
sentencias
or
delay expresión;
sentencias
end select;

Veamos algunos ejemplos de las instrucciones anteriores:

```
- Intrucciones de control
     if MINUTO = 12 and SEGUNDO = 60
          then
          MINUTO:=13
```

```
end if;

case DIASEMANA is
     when Ø => return DOMINGO;
     when 1 => return LUNES;
     when 2 => return MARTES;
     when 3 => return MIERCOLES;
end case;
SUMA:= Ø

for VALOR in  1..5 loop
SUMA:= SUMA + VALOR;
end loop;

loop
LEER (CARACTER);
exit when CARACTER = '*';
end loop;
```

Programa 1. Ejemplo de instrucciones de control.

```
- Bloque
          INTERCAMBIO;
          declare
               TEMPORAL :INTEGER;
          begin
               TEMPORAL:=V; V:=U;
               U:=TEMPORAL;
          end INTERCAMBIO;
```

Programa 2. Ejemplo de bloque.

En este ejemplo se pone de manifiesto que la noción de bloque nos permite introducir una nueva región en la parte ejecutable de una unidad de programa.

```
- Instrucción return
     return Z ** 2;      (en una función)
     return MAXIMO = MINIMO;   (en una función)
     return;   (en un procedimiento)

- Instrucción goto
     if X(I) > ELEMENTO then
          if DERECHA(I) /= Ø then
               I:=DERECHA(I);
               goto COMPARAR;
          end if;
          ..........
          .........
          end if;

- Instrucción raise
     raise ERROR_NUMERICO; produce una excepción predefinida
- Llamada a procedimiento
 procedimiento  NUMEROS  (IZQ,DER  :  NOMBRE_NUMERO  :=  new
                                   NUMERO)
 -- posibles llamadas
     NUMEROS;
     NUMEROS (IZQ => new NUMERO, DER => new NUMERO);
```

Programa 3. Ejemplos de goto, return y raise.

INSTRUCCIONES ORIENTADAS A TAREAS

Las instrucciones *accept* definen las acciones que se producirán cuando otra tarea llame a la entrada indicada.

```
          accept E(parámetros formales) do
               cuerpo de E
           end E;

     Una implementación del tipo :
          loop
               accept DEPOSITAR (msg : in mensaje) do
```

```
        ............
        end DEPOSITAR;
        accept SACAR (msg : out mensaje) do
        ...........
        end SACAR;
   end loop;
```

Programa 4. Ejemplo de instrucciones orientadas a tareas.

Especifica que DEPOSITAR y SACAR se ejecutan alternativamente. A cada DEPOSITAR le sigue un SACAR, y cada ejecución de SACAR va seguida por un DEPOSITAR.

Otra instrucción orientada a tarea es la *delay*

```
delay 4.Ø    -- retardo de 4.Ø segundos
declare
use HORARIO;
 SIGUIENTE : TIEMPO := RELOJ + INTERVALO;
begin
     loop
          delay SIGUIENTE - RELOJ;
          .............
          SIGUIENTE := SIGUIENTE + INTERVALO;
     end loop;
end;
```

Programa 5. Ejemplo de la instrucción DELAY.

La instrucción *select* permite escribir una tarea de forma que puedan aceptarse «entradas» (entry call).

```
loop
 select
  accept DEPOSITAR (msg : in mensaje) do
     ...........
  end DEPOSITAR;
```

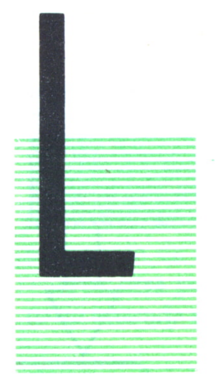
```
        or
                accept SACAR (msg : out mensaje) do
                ...........
                end SACAR;
            end select;
        end loop;
```

Programa 6. Ejemplo de «select».

El efecto de esta instrucción es que tanto DEPOSITAR como SACAR se ejecutan de forma que ambas se comportan como secciones críticas una con respecto a la otra.

TAREAS Y TRATAMIENTO DE EXCEPCIONES 8

LAS TAREAS Y EL CONCEPTO DEL < <RENDEZVOUS> >



AS tareas representan el conjunto más complejo de características del lenguaje Ada. Resumiremos, a grandes rasgos, algunas de las características más importantes de las tareas:

1. Las tareas representan líneas paralelas del programa de control que «correrán» concurrentemente en tiempo real sobre un multiprocesador o en tiempo real (aparente) sobre un simple procesador.
2. Toda la planificación, en tiempo de ejecución, de las tareas se gestiona por el <<entorno>> cuyas características e implementación son transparentes (ocultas) al programador.
3. La activación de una tarea se controla a través de la localización de la declaración de la tarea dentro del programa y el tipo de declaración asociada con la tarea.
4. La interacción, comunicación y datos compartidos entre tareas se lleva a cabo a través de un mecanismo conocido por *rendezvous*.
5. Las tareas son unidades de programa, así como tipos de objetos.
6. Se pueden declarar tipos de tareas y objetos de ese tipo, los cuales pueden ser considerados como un objeto. Esto significa que objetos del tipo tarea pueden ser pasados como parámetros en llamadas a procedimientos, declarados como componentes de un array y apuntados por tipos acceso (punteros).

Cuando la tarea que «llama» y la tarea «llamada» tienen su llegada en un punto donde ambas esperan que se produzca la comunicación, este punto se denomina un *rendezvous*.

La sincronización entre tareas se consigue por la llamada a una entrada, en una tarea especificada, y respondiendo a ella dentro de la tarea llamada con una instrucción *accept*. Al final de la instrucción *accept* se ha completado el *rendezvous*.

Un *rendezvous* ocurre cuando una tarea «expresa» su deseo de llamar

a un punto de entrada (entry point) dentro de otra tarea y esa otra tarea acepta la llamada de entrada. Múltiples llamadas a una entrada son puestas en cola para un tratamiento secuencial.

El mecanismo del *rendezvous* podemos describirlo de la siguiente manera:

— Los parámetros actuales *in* e *in out* del punto de entrada se copian dentro de los parámetros formales de la instrucción *accept*.

— Las sentencias internas a la instrucción *accept* son ejecutadas.

— Los parámetros formales *in out* y *out* de la instrucción *accept* se copian de nuevo en los parámetros actuales del punto de entrada, completando la ejecución.

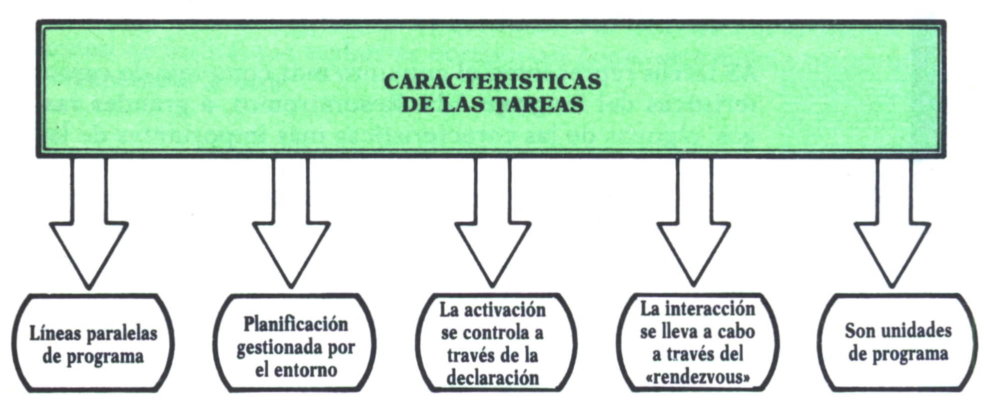


Fig. 1. Características de las tareas.

Cuando una tarea quiere retardarse puede emplearse la instrucción *delay*.

delay expresión;

donde expresión es una simple expresión de tipo «duración» expresada en segundos.

El incremento más pequeño de «duración» no debe ser superior a 20 milisegundos.

Puntos de entrada con retardo permiten a las tareas desarrollar algunas otras acciones si su punto de entrada no es aceptado en un momento determinado.

select
instrucción de punto de entrada;
secuencia de instrucciones
or
delay expresión de retardo;
secuencia de instrucciones
end select;

Si el rendezvous puede comenzar dentro del retardo especificado aquél comienza y la primera secuencia de instrucciones se ejecuta. Por otra parte, el punto de entrada es cancelado después de que el retardo especificado haya expirado, ejecutándose la segunda secuencia de instrucciones. La cancelación del punto de entrada lleva consigo la eliminación de la llamada de la cola de entrada.

Los puntos de entrada (condicionales) permiten a las tareas desarrollar otras acciones si en el punto de entrada no se produce un rendezvous.

select
punto de entrada;
sentencias;
else
sentencias;
end select;

Si la tarea llamada puede establecer un *rendezvous* con la tarea llamante, de forma inmediata, entonces el *rendezvous* se producirá y la primera secuencia de instrucciones se ejecutará.

Por otra parte, si el *rendezvous* no puede producirse, entonces el punto de entrada se cancela y la segunda secuencia de instrucciones se ejecuta.



TRATAMIENTO DE EXCEPCIONES

Ada proporciona un potente mecanismo para definir las condiciones excepcionales que puedan ocurrir durante la ejecución de un programa. Algunas excepciones están ya predefinidas, pero otras pueden ser definidas por el usuario. Las excepciones definidas por el usuario permiten especificar las condiciones de error o excepcionales que son de capital importancia para el programa.

El lenguaje Ada posee, en relación a las excepciones, las siguientes características:

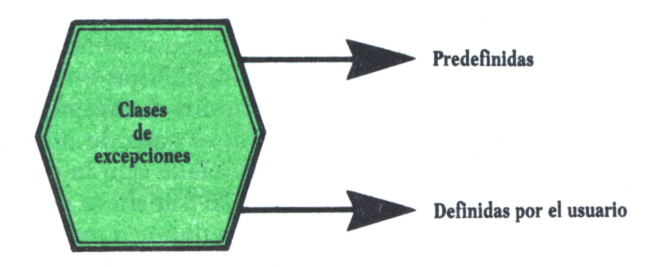


Fig. 2. Clases de excepciones.

Excepciones predefinidas

Condiciones de error definidas en el lenguaje para detectar violaciones en los tipos de datos, errores de procesamiento, etc.

Excepciones definidas por el usuario

Es una facilidad para la declaración de excepciones.

Instrucción raise

Permite generar una condición excepcional que detiene la ejecución normal del programa y transfiere control a una secuencia de instrucciones que hacen un tratamiento de la excepción.

Reglas para la propagación de excepciones

Las reglas del lenguaje que rigen la forma en la cual las excepciones se propagan de un subprograma o tarea a otro.

Manipulador de excepciones

Es un área, dentro de una unidad de programa, donde la secuencia de instrucciones que lleva a cabo el tratamiento puede aparecer.

Veamos un ejemplo de tratamiento de excepciones:

```
begin
.............
exception
when ERROR DATOS => null;
raise ERROR;
end;
```

Cuando se produce una excepción, la ejecución normal del programa se suspende. Si existe un «manipulador» de excepciones en el interior del cuerpo del subprograma, bloque de instrucciones o cuerpo del paquete, el control se transfiere a ese manipulador. En el caso de no encontrarse manipuladores en la secuencia de llamada asociada con la ocurrencia, la ejecución del programa se detiene y el control se transfiere de nuevo al sistema operativo. Cuando se encuentra un manipulador o manejador, la ejecución de éste origina la ejecución del bloque que contiene al manejador y a partir de aquí el flujo normal del programa continúa.

Las excepciones modificarán la secuencia normal de un programa.

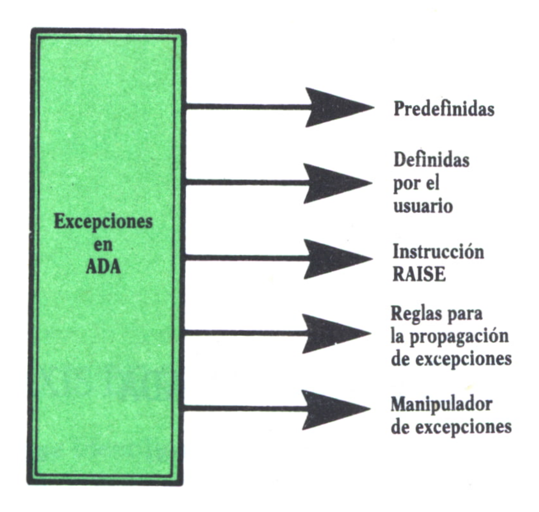


Fig. 3. Excepciones en ADA.

```
function FACTORIAL (A:INTEGER) return FLOAT is
     begin
          if A=1 then
               return 1;
          else
          return FLOAT(A) * FACTORIAL(A - 1);
          end if;
     exception
          when NUMERIC_ERROR => return FLOAT'SAFE_LARGE;
   end FACTORIAL;
```

Programa 1. Ejemplo de excepciones.

APENDICE A



PALABRAS RESERVADAS EN ADA

abort	*abs*	*accept*	*access*	*all*
and	*array*	*at*	*begin*	*body*
case	*constant*	*declare*	*delay*	*delta*
digits	*do*	*else*	*elsif*	*end*
entry	*exception*	*exit*	*for*	*function*
generic	*goto*	*if*	*in*	*is*
limited	*loop*	*mod*	*new*	*not*
null	*of*	*or*	*others*	*out*
package	*pragma*	*private*	*procedure*	*raise*
range	*record*	*rem*	*renames*	*return*
reverse	*select*	*separate*	*subtype*	*task*
terminate	*then*	*type*	*use*	*when*
while	*with*			

APENDICE B



SINTAXIS DE LOS TIPOS DE DATOS

type identificador *is* definición de tipo

* *tipos escalares*

tipos enumerados

type nombre del tipo *is* (identificador);

tipos carácter

type nombre del tipo *is* (literal);

tipos enteros

type nombre del tipo *is* límite inf ... límite sup;

tipos reales

— *En punto flotante*

type nombre del tipo is digits *número* de dígitos;

— *En punto fijo*

type nombre del tipo is delta precisión

* *tipos compuestos*

tipos array

— *restringidos*

type nombre del tipo *is array* (margen) of subtipo;

— *no restringidos*

type nombre del tipo *is array* (marca *range* <>) *of* subtipo;

tipos registro

type nombre del tipo *is*
record
    lista de componentes
end record;

* *tipos acceso (punteros)*

type nombre del tipo *is access* subtipo apuntado;

* *tipos privados*

type nombre del tipo *is [limited] private;*

APENDICE C



PRAGMAS DEFINIDOS EN EL LENGUAJE ADA

CONTROLLED

Especifica que una petición de almacenamiento automático debe generarse para el objeto apuntado por este tipo acceso (puntero).

ELABORATE

INLINE

INTERFACE

Especifica que el subprograma se codifica en otro lenguaje y se proporciona como código objeto.

LIST

MEMORY SIZE

Especifica el valor utilizado para el número denominado SYSTEM.MEMORY_SIZE, el cual define el número de unidades de almacenamiento en la configuración de la máquina («target machine»).

OPTIMIZE

Especifica si el tiempo o el espacio se utilizarán con criterios de optimización.

PACK

De utilidad para empaquetamiento en memoria.

PAGE

Especifica que el texto del programa comenzará en una nueva página.

PRIORITY

Especifica la prioridad de una tarea o subprograma.

SHARED

Especifica que cada lectura o actualización de la variable es un punto de sincronización para esa variable.

STORAGE_SIZE

Especifica el valor que se utiliza para el número denominado SYSTEM.STORAGE_SIZE el cual define el número de bits por unidad de almacenamiento en la configuración de la máquina.

SUPPRESS

Toma como argumento, opcionalmente, el nombre de un objeto, tipo (subtipo), subprograma, tarea o genérico.

SYSTEM_NAME

Especifica el valor que se utiliza para la constante SYSTEM.SYSTEM_NAME.

APENDICE D



OPERACIONES Y ATRIBUTOS EN ADA

** OPERACIONES*

Asignación

:=

Asignador

new

Prueba entre miembros

in
not in

Control de ruptura

and then
or else

Componente seleccionado

nombre1.nombre2

Componente indexado

nombre(exp1)

** OPERADORES*

Lógicos

and or xor

Relacionales

= /= < <= > >=

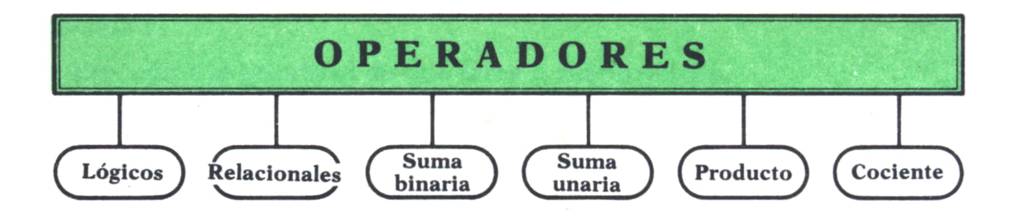


Fig. 1. Operadores en Ada.

Suma binaria

+ – &

Suma unaria

+ –

Producto

*

Cociente

/ mod rem

Orden de precedencia

*** abs not*

* *ATRIBUTOS DE UTILIDAD*

'FIRST

Límite inferior de un tipo discreto.
Límite inferior (índice 1) de un array.

'FIRST(N)

Límite inferior (índice N) de un array.

'LAST

Límite superior de un tipo discreto.

Límite superior (índice 1) de un array.

'LAST(N)

Límite superior (índice N) de un array.

'LENGTH

Longitud (índice 1) de un array u objeto.

'LENGTH(n)

Longitud (índice N) de un array.

POS(X)

Posición numérica de X dentro de la declaración tipo.

PRED(X)

Predecesor de X para este tipo.

'RANGE

Margen (índice 1) para un array u objeto.

'RANGE(N)

Margen (índice N) para un array u objeto.

'SUCC(X)

Siguiente a X para este tipo.

'VAL(X)

Es una función de conversión.

* *OTROS ATRIBUTOS*

Aft, Fore, Image, Last_Bit, Mantissa, Position, Value, Width, Address, Base, Constrained, Size, Storage_Size, Delta, Digits, Emax, Epsilon, Large, Safe_Emax, Safe_Large, Safe_Small, Small, Callable, Count, Storage_Size, Terminated, Machine_Emax, Machine_Emin, Machine_Mantissa, Machine_Overflows, Machine_Radix, Machine_Rounds.

APENDICE E



UNIDADES DE LIBRERIA DEFINIDAS EN EL LENGUAJE

STANDARD

Define todas las excepciones predefinidas (excepto I/O).
Define todas las operaciones y tipo de datos predefinidos.
Es el paquete dentro del cual las otras unidades de librería se compilan.

CALENDAR

Define tipos de datos y operaciones sobre el tipo TIME.
Utilizada en «tareas» y aplicaciones que precisan acceso a tiempo real.

SYSTEM

Define aspectos de implementación de un sistema Ada específico.

MACHINE_CODE

Define la «interface» a nivel de código máquina para la máquina objeto (target machine).

UNCHECKED_DEALLOCATION

Permite al usuario controlar las asignaciones de los objetos tipo acceso.

UNCHECKED_CONVERSION

Permite conversiones a nivel de bits.

SEQUENTIAL_IO

Proporciona operaciones de E/S a cinta.

DIRECT_IO

Proporciona operaciones de E/S a disco.

TEXT_IO

Proporciona operaciones de E/S legibles por el usuario.

IO_EXCEPTIONS

Define excepciones de E/S.

LOW_LEVEL_IO

Define operaciones de acceso a bajo nivel para dispositivos de E/S.

APENDICE F



EXCEPCIONES DEFINIDAS EN EL LENGUAJE ADA

Constraint_Error

Las condiciones en que se alcanza esta excepción son:

* Intentar asignar a un objeto un valor fuera de margen.
* El valor de un índice está fuera de los límites.
* Operaciones lógicas sobre arrays de diferentes longitudes.

Data_Error

Las condiciones en que se da esta excepción son:

* Cuando en una operación de E/S se intenta leer un elemento que no es del tipo correcto o está dentro del margen permitido de valores.

Device_Error

Las condiciones en que se da esta excepción son:

* Cuando existe un mal funcionamiento del hardware de E/S.

End_Error

Las condiciones en que se da esta excepción son:

* Cuando se intenta leer más allá del fin de fichero.

Layout_Error

Numeric_Error

Las condiciones en que se da esta excepción son:

* Ejecución de una operación numérica predefinida que no puede dar el resultado correcto (un desbordamiento, por ejemplo).

Program_Error

Las condiciones en que se da esta excepción son:

* Llamar a un subprograma cuyo «cuerpo» no ha sido desarrollado.
* Intentar salir de una función con una instrucción distinta a «return».
* Otros errores de programa.

Storage_Error

Las condiciones en que se da esta excepción son:

* Insuficiente capacidad de memoria para guardar nuevos objetos o activar nuevas unidades de programa.

Tasking_Error

Las condiciones en que se da esta excepción son:

* Cuando una tarea falla durante su activación.
* En un punto de entrada a una tarea anormal o completada.

APENDICE G



GLOSARIO DE TERMINOS

Acoplamiento

Medida de la independencia entre módulos de un programa.

Abstracción de datos

Resultado de extraer y retener sólo las características esenciales de los datos.

Asignador

La evaluación de un asignador crea un objeto y devuelve un nuevo tipo acceso (puntero) el cual apunta al objeto.

Cola

Una lista a la que se accede de la forma «primero en llegar, primero en salir».

Componente

Valor que forma parte de otro más amplio o un objeto que forma parte de otro superior.

Constante

Un objeto cuyo valor no cambia una vez inicializado.

Data

Representación formal de hechos, conceptos o instrucciones que sirven para comunicación, interpretación o procesamiento por medios humanos o automáticos.

Encapsulación

La técnica de aislar una función dentro de un módulo y proporcionar una especificación precisa para dicho módulo.

Entrada

Una entrada (entry) se utiliza para comunicación entre tareas.

Excepción

Es una situación errónea a la cual se llega durante la ejecución de un programa. Alcanzar una excepción supone abandonar la ejecución normal del programa.

Función

Especifica una secuencia de instrucciones y devuelve un valor llamado resultado.

Manipulador (Handler)

Una parte de un programa que especifica una respuesta a una excepción.

Interfase (Interface)

Un límite compartido.

Interrupción

Suspensión de un proceso,como podría ser la ejecución de un programa de ordenador, causado por un suceso externo al proceso.

Objeto

Un objeto contiene un valor. Un programa crea un objeto bien por elaboración de una declaración objeto, bien evaluando un asignador.

Operador

Un operador es una operación que tiene uno o dos operandos.

Paquete

Especifica un grupo de entidades relacionadas lógicamente, tales como tipos, objetos de esos tipos y subprogramas con parámetros de aquellos tipos.

Pragma

Suministra información al compilador.

Parte privada

Parte de la especificación de un paquete que contiene detalles estructurales que completan la especificación de las entidades visibles.

Parámetro actual

Una entidad particular asociada con el correspondiente parámetro formal a través de una llamada a subprograma, punto de entrada (entry call), etcétera.

Rango

Conjunto continuo de valores de tipo escalar.

Rendezvous

Interacción que sucede entre dos tareas cuando una tarea llamó a la entrada (entry) de la otra tarea y una instrucción de aceptación se ejecuta por tarea llamada.

Reusabilidad

Propiedad por la cual un módulo puede ser utilizado en múltiples aplicaciones.

Software de aplicación

Software específicamente desarrollado para utilización funcional de un sistema informático.

Stub

Un módulo de programa ficticio, utilizado durante el desarrollo y pruebas de un módulo de alto nivel.

Tarea

Opera en paralelo con otras partes del programa.

Validación

Proceso de evaluación del software que pretende asegurar al final del desarrollo el cumplimiento de aquél con los requisitos establecidos.

Variable

Objeto que contiene un valor que puede cambiar durante la ejecución del programa.

APENDICE H

Tabla ASCII

CONVERSION NUMERICA

CONVERSIONES NUMERICAS DECIMAL-HEXADECIMAL-OCTAL-BINARIO-ASCII

DEX X_{10}	HEX X_{16}	OCT X_8	BINARIO P X_X	ASCII	Tecla *
0	00	00	0 000 0000	NUL	CTRL/1
1	01	01	1 000 0001	SOH	CTRL/A
2	02	02	1 000 0010	STX	CTRL/B
3	03	03	0 000 0011	ETX	CTRL/C
4	04	04	1 000 0100	EOT	CTRL/D
5	05	05	0 000 0101	ENQ	CTRL/E
6	06	06	0 000 0110	ACK	CTRL/F
7	07	07	1 000 0111	BEL	CTRL/G
8	08	10	1 000 1000	BS	CTRL/H, RETROCESO
9	09	11	0 000 1001	HT	CTRL/I, TAB
10	0A	12	0 000 1010	LF	CTRL/J, SALTO LINEA
11	0B	13	1 000 1011	VT	CTRL/K
12	0C	14	0 000 1100	FF	CTRL/L
13	0D	15	1 000 1101	CR	CTRL/M, RETURN
14	0E	16	1 000 1110	SO	CTRL/N
15	0F	17	0 000 1111	SI	CTRL/O
16	10	20	1 001 0000	DLE	CTRL/P
17	11	21	0 001 0001	DC1	CTRL/Q
18	12	22	0 001 0010	DC2	CTRL/R
19	13	23	1 001 0011	DC3	CTRL/S
20	14	24	0 001 0100	DC4	CTRL/T
21	15	25	1 001 0101	NAK	CTRL/U
22	16	26	1 001 0110	SYN	CTRL/V
23	17	27	0 001 0111	ETB	CTRL/W

DEX X_{10}	HEX X_{16}	OCT X_{8}	BINARIO P X_{X}	ASCII	Tecla *
24	18	30	0 001 1000	CAM	CTRL/X
25	19	31	1 001 1001	EM	CTRL/Y
26	1A	32	1 001 1010	SIB	CTRL/Z
27	1B	33	0 001 1011	ESC	ESC, ESCAPE
28	1C	34	1 001 1100	FS	CTRL <
29	1D	35	0 001 1101	GS	CTRL/
30	1E	36	0 001 1110	RS	CTRL/=
31	1F	37	1 001 1111	US	CTRL/-
32	20	40	1 010 0000	SP	ESPACIO
33	21	41	0 010 0001	!	!
34	22	42	0 010 0010	”	”
35	23	43	1 010 0011	#	#
36	24	44	0 010 0100	$	$
37	25	45	1 010 0101	%	%
38	26	46	1 010 0110	&	&
39	27	47	0 010 0111	‘	‘
40	28	50	0 010 1000	(	(
41	29	51	1 010 1001	)	)
42	2A	52	1 010 1010	*	*
43	2B	53	0 010 1011	+	+
44	2C	54	1 010 1100	’	’
45	2D	55	0 010 1101	–	–
46	2E	56	0 010 1110	.	.
47	2F	57	1 010 1111	/	/
48	30	60	0 011 0000	0	0
49	31	61	1 011 0001	1	1
50	32	62	1 011 0010	2	2
51	33	63	0 011 0011	3	3
52	34	64	1 011 0100	4	4
53	35	65	0 011 0101	5	5
54	36	66	0 011 0110	6	6
55	37	67	1 011 0111	7	7
56	38	70	1 011 1000	8	8
57	39	71	0 011 1001	9	9
58	3A	72	0 011 1010	:	:
59	3B	73	1 011 1011	;	;
60	3C	74	0 011 1100	<	<
61	3D	75	1 011 1101	=	=
62	3E	76	1 011 1110	>	>
63	3F	77	0 011 1111	?	?
64	40	100	1 100 0000		
65	41	101	0 100 0001	A	A
66	42	102	0 100 0010	B	B
67	43	103	1 100 0011	C	C
68	44	104	0 100 0100	D	D

DEX X_{10}	HEX X_{16}	OCT X_8	BINARIO $P\ X_X$	ASCII	Tecla *
69	45	105	1 100 0101	E	E
70	46	106	1 100 0110	F	F
71	47	107	0 100 0111	G	G
72	48	110	0 100 1000	H	H
73	49	111	1 100 1001	I	I
74	4A	112	1 100 1010	J	J
75	4B	113	0 100 1011	K	K
76	4C	114	1 100 1100	L	L
77	4D	115	0 100 1101	M	M
78	4E	116	0 100 1110	N	N
79	4F	117	1 100 1111	O	O
80	50	120	0 101 0000	P	P
81	51	121	1 101 0001	Q	Q
82	52	122	1 101 0010	R	R
83	53	123	0 101 0011	S	S
84	54	124	1 101 0100	T	T
85	55	125	0 101 0101	U	U
86	56	126	0 101 0110	V	V
87	57	127	1 101 0111	W	W
88	58	130	1 101 1000	X	X
89	59	131	0 101 1001	Y	Y
90	5A	132	0 101 1010	Z	Z
91	5B	133	1 101 1011	[	[
92	5C	134	0 101 1100	\	\
93	5D	135	1 101 1101	]	]
94	5E	136	1 101 1110	ˆ	ˆ
95	5F	137	0 101 1111	-	-
96	60	140	0 110 0000	`	`
97	61	141	1 110 0001	a	a
98	62	142	1 110 0010	b	b
99	63	143	0 110 0011	c	c
100	64	144	1 110 0100	d	d
101	65	145	0 110 0101	e	e
102	66	146	0 110 0110	f	f
103	67	147	1 110 0111	g	g
104	68	150	1 110 1000	h	h
105	69	151	0 110 1001	i	i
106	6A	152	0 110 1010	j	j
107	6B	153	1 110 1011	k	k
108	6C	154	0 110 1100	l	l
109	6D	155	1 110 1101	m	m
110	6E	156	1 110 1110	n	n
111	6F	157	0 110 1111	o	o
112	70	160	1 111 0000	p	p
113	71	161	0 111 0001	q	q

DEX X_{10}	HEX X_{16}	OCT X_8	BINARIO P X_X	ASCII	Tecla *
114	72	162	0 111 0010	r	r
115	73	163	1 111 0011	s	s
116	74	164	0 111 0100	t	t
117	75	165	1 111 0101	u	u
118	76	166	1 111 0110	v	v
119	77	167	0 111 0111	w	w
120	78	170	0 111 1000	x	x
121	79	171	1 111 1001	y	y
122	7A	172	1 111 1010	z	z
123	7B	173	0 111 1011	{	{
124	7C	174	1 111 1100	¦	¦
125	7D	175	0 111 1101	}	}
126	7E	176	0 111 1110	~	~
127	7F	177	1 111 1111	DEL	DEL, BORRAR

APENDICE I

BIBLIOGRAFIA

Practitioner's Guide to Ada. Robert H. Wallace.

Ada for experienced programmers. A. Nico Habermann. Deway-ne E. Perry.

Self reproducing Ada tasks. Nico Lomuto.

Using Ada for commercial software. Michael R. Gardner.

El considerable esfuerzo desarrollado por el Departamento de Defensa de los EE.UU. (DoD), para que el lenguaje Ada fuese desarrollado quedará compensado por las aportaciones de este lenguaje y su entorno a los sistemas informáticos del futuro.

En Ada se conjugan el entorno y el lenguaje para dar como resultado un gran soporte a la ingeniería del software.

Sus aplicaciones originales, sistemas en tiempo real para mando y control en el área de la Defensa, han sido ampliadas al campo industrial para el control de procesos, inteligencia artificial, etc.

Muchos de los conceptos de la ingeniería del software tales como modularidad, fiabilidad, transportabilidad, etc., quedan recogidos en el ya conocido como lenguaje de la década de los noventa.